Sustainable Computing: Informatics and Systems 3 (2013) 231–248

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Dynamic power management for QoS-aware applications

Moreno Marzolla[a,*], Raffaela Mirandola[b]

[a] *Università di Bologna, Dipartimento di Scienze dell'Informazione, Mura A. Zamboni 7, I-40126 Bologna, Italy*
[b] *Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci, I-20133 Milano, Italy*

A R T I C L E I N F O

Article history:
Received 22 June 2011
Received in revised form 20 June 2012
Accepted 26 February 2013

Keywords:
Power management
Performance modeling
SLA management
Energy proportional computing

A B S T R A C T

Reducing the power requirement of large IT infrastructures is becoming a major concern. Energy savings can be achieved with hardware and/or software solutions; in particular, modern CPUs can operate at different power levels that can be selected by software: low power modes reduce energy consumption at the cost of lowering also the CPU processing rate. In this paper we address the problem of reducing energy consumption of a large-scale distributed application subject to Service Level Agreements requiring a maximum allowed response time. Specifically, we propose *Energy Aware reconfiguration of software SYstems* (EASY), an on-line algorithm for dynamically adjusting the processing speed of individual devices such that the average system response time is kept below a predefined threshold, and the total power consumption is minimized. EASY uses a queueing networks performance model to proactively drive the reconfiguration process, so that the number of individual reconfiguration actions is reduced. We formulate the energy conservation problem as a Mixed Integer Programming problem, for which we propose a heuristic solution technique. Numerical experiments show that the heuristic produces almost optimal results at a substantially lower computational cost. Therefore, EASY can be effectively applied on-line to make a large system energy-proportional.

1. Introduction

The constant growth of energy usage in industrialized countries is creating problems to the sustainability of the Earth development. According to a report by Koomey [1], the total electrical power consumed by servers worldwide doubled from 2000 to 2005, representing an annual growth rate of 14% yearly for the U.S. and 16% worldwide. The total power consumption of all servers in the U.S. in 2005 was about 2.6×10^{12} W (2.6 billion kW); all servers in the world are estimated to consume about 7×10^{12} W. If we include also the energy required for cooling and for powering ancillary equipment, the total power consumption raises to 5.2×10^{12} W for the U.S., and 14×10^{12} W worldwide [1, Table 6].

The interest towards efficient use of technology is also motivated by some alarming trends showing, for example, that computing equipment in the U.S. alone is estimated to consume more than 20 million giga-joules of energy per year, the equivalent of four-million tons of carbon-dioxide emissions into the atmosphere [2,3]. IT analysis firm IDC[1] estimates the total worldwide spending on power management for enterprises was likely 40 billion dollars in 2009.

For the reasons above, the *green computing* discipline addresses the problem of building and managing computing infrastructures that provide the same quality of service with lower energy consumption [3]. Rudimentary techniques for power management, e.g., shutting down idle servers, can impact the ability of the hosting center to meet the Service Level Agreements (SLAs) implicitly or explicitly stipulated with clients. Shutting down servers ensures the maximum power saving; however, bringing up a machine when needed requires a significant start-up time (up to several minutes, depending also on the time needed to start the applications). Start-up delays can severely impact the ability of the system to promptly handle workload fluctuations. Besides, repeated on-off cycles can stress hardware components, increasing the probability of failures, which add further costs for repair or replacement of broken devices [4].

One important observation is that the average utilization of a server in a datacenter is quite low, but rarely zero: Barroso and Hölzle observe that most of the time, servers operate between 10% and 50% of their maximum utilization level [5], and the utilization rarely drops to zero. This behavior is not accidental, but derives from the application of appropriate engineering and provisioning principles: if the number of servers is reduced to increase their average utilization, then the system would operate near its saturation point. In Fig. 1 we qualitatively show the dependency of response time from utilization of a generic service center (being it a single device or a whole datacenter): as the utilization approaches one, the response time diverges since an infinite queue of requests

* Corresponding author. Tel.: +39 051 2094847; fax: +39 051 2094510.
E-mail addresses: marzolla@cs.unibo.it (M. Marzolla), mirandola@elet.polimi.it (R. Mirandola).

[1] http://www.idc.com/.

2210-5379/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.suscom.2013.02.001

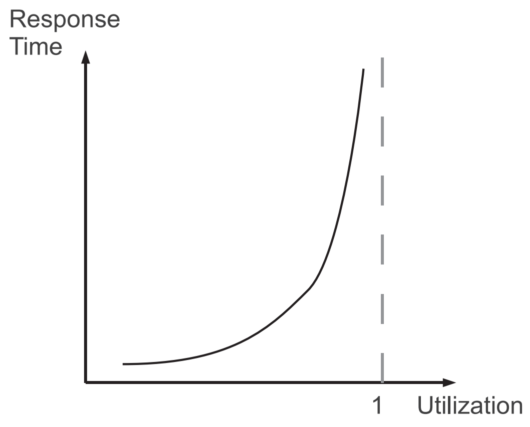

Fig. 1. The response time of a generic service center (being it a single device or a whole datacenter) response time diverges as its utilization approaches one, as shown qualitatively in this picture.

builds up; in practice, overloaded devices drop requests as internal buffers fill up. A certain amount of slack capacity is necessary to avoid operating the system near the saturation point, to cope with workload fluctuations.

We also observe that an important family of large scale Web applications includes the so called Online Data Intensive (OLDI) services [6] such as search engines, online advertising, Massive Multiplayer Online Games (MMOGs) and similar applications that process external requests under very strict response time constraints. OLDI workloads are driven by user-generated queries that interact with massive data sets; responses must be produced within a very short time, sometimes in the sub-second scale. Furthermore, OLDI workloads fluctuate considerably over time, usually with a periodic pattern resulting from human activity periods.

As an example, we show in Fig. 2 the number of online players connected to the RuneScape MMOG, collected by monitoring the game Web page during several weeks starting from April 2011 (more details will be given in Section 9). A daily pattern is clearly visible, which results from the overlapping of activity periods of a large population of players distributed over different time zones.

The discussion above suggests to reduce the power consumption by making the system *power proportional* [5], meaning that the power consumption of devices is kept proportional to their utilization. Ideally, a device should consume zero power when its utilization is zero, and full power when its utilization is one. Unfortunately, such "perfect" power proportionality has not been achieved yet, current server processors can consume about 30% of their peak power during very low activity periods, creating a dynamic range of more than 70 % their peak power. CPUs targeted at mobile devices have idle power usually reaching one tenth of their peak power.

Specifically, most processors allow dynamic frequency/voltage scaling (DVS), where the frequency (and voltage) can be lowered to produce much more savings in power consumption compared to how much one looses on quality. The Advanced Configuration and Power Interface (ACPI) specification has been proposed as an open standard for configuration and power management of individual devices or entire systems [7]. ACPI defines a larger set of low power states, some of which are fully operational, meaning that the device still processes user requests at a possibly slower rate. For processors, power savings within operational states is achieved through *dynamic frequency scaling*, in which the CPU frequency and voltage are dynamically adjusted to reduce the energy requirement. Since ACPI state changes can be controlled by software, there is a great potential for achieving significant energy reductions by implementing more elaborate power/performance tradeoffs, which were not possible with the previous generation of devices.

1.1. Contribution of this paper

In this paper we pursue this direction by proposing a dynamic power management strategy that minimizes the overall power consumption of a large scale IT infrastructure subject to quality of service (QoS) constraints in the form of a maximum allowed response time. While using dynamic voltage scaling may not provide as much savings as shutting down a server completely, the advantage is that it can still service requests (albeit at a lower frequency), and it can be switched back to full speed with negligible overhead. This is essential for OLDI applications mentioned above.

We propose the Energy Aware reconfiguration of software SYstems (EASY) framework for dynamic power optimization of large-scale systems subject to response time constraints. Specifically, the system response time must be kept below a given threshold. The system is made of a set of ACPI-capable devices (CPUs, disks, system modules and so on). We assume that the system is subject to variable workload; the internal load of the system can be unevenly spread across the servers, and can also shift from device to device at any time. EASY is an on-line selection of power/performance levels of devices such that the average response time is kept below the threshold, and the total energy consumption rate is minimized. EASY can be applied to any device supporting the ACPI specification (not just CPUs), hence our approach is quite general and can be applied to many different kind of systems.

EASY is based on a Monitor-Analyze-Plan-Execute control loop [8]: the system is enhanced with a monitor component, which is responsible for collecting performance measures at runtime. This information is used both to identify when the response time constraint is being violated, and also to drive the reconfiguration process itself. The analyzer and the planner use a simple queueing network (QN) model to quickly analyze different power/performance settings. All quantitative parameters needed to evaluate the QN model are estimated by the monitor: hence, EASY does not require any prior knowledge on the system. Additionally, EASY can adapt to both changes in the workload, or changes in the load distribution within the devices. The allowed maximum system response time can be changed at run time if necessary, allowing system administrators to easily redefine the SLA. We formulate the energy minimization problem as a Mixed Integer Programming (MIP) problem. Besides, we describe an effective heuristic that can provide a feasible solution faster than handling the optimization problem with a conventional general-purpose solver. Using a set of simulation experiments, we show that the heuristic used by EASY is very fast, and the quality of the results is

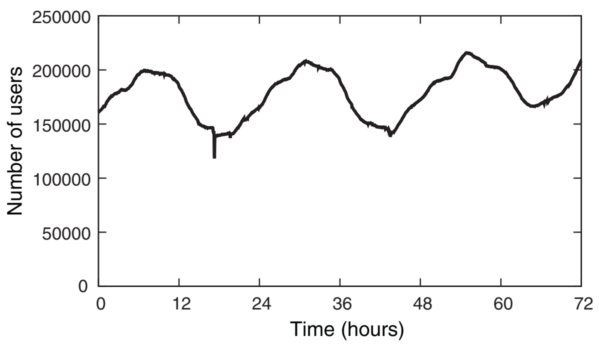

Fig. 2. Number of RuneScape players.

statistically comparable to the optimal solution of the MIP problem computed through a MIP solver.

1.2. Paper organization

The remainder of the paper is organized as follows. In Section 2 we review the scientific literature and compare existing works with our approach. In Section 3 we describe the ACPI specification. Section 4 formally describes the problem which we are addressing. The high level architecture of EASY is described in Section 5; Section 6 describes the QN performance model, and how response times can be estimated from the quantitative parameters collected by the monitor. In Section 7 we formulate the optimization problem as a MIP problem, which can be handled by standard MIP solvers. In Section 8 we describe an effective heuristic which can be used to solve the same problem much more efficiently. The results of the extensive simulation experiments used to validate our approach are then presented and discussed in Section 9. Finally, Section 10 presents concluding remarks, and proposes future research directions along which the approach described in this paper can be extended. To make this paper self-contained, we include in Appendix: a table summarizing the main notation used in the paper, a short overview of queueing network modeling, and a description of the MIP problem using the GNU MathProg language.

2. Related work

Adaptive computing systems have been studied extensively over the last years, by several research communities from different point of views [9]. The *autonomic computing* framework is a notable example of a general approach to the design of adaptive computing systems [8,10]. This paper deals with model-driven self adaptation of systems subject to response time constraints, where the goal of the adaptation strategy is to minimize the overall energy consumption. Therefore, our review of the scientific literature will focus on works proposing trade-off between energy consumption and performance.

A recent trend in managing large datacenters is the increasing reliance on virtualization technologies. Virtualization is one of the key enabler of the Cloud Computing paradigm [11], and can be applied at various levels of the IT infrastructure (processor, storage, network, etc.). At the processor level, modern CPUs support the execution of (isolated) guest Operating Systems within the same host OS.

Virtualization has been exploited to reduce the power consumption of the host machine. In [12] the authors propose VirtualPower, a mechanism for reflecting the power management policies of the guest OSes to the real hardware. VirtualPower allows the host OS to take actions according to the guest VM independent energy management policies; this enables the host OS to attain global objectives based on knowledge about guest OSes. In [13] the authors propose a Peer-to-Peer algorithm for consolidating virtual machine instances using live migration, with the aim of reducing the number of physical servers that need to be powered on at a given time. In [14] the authors propose a more general consolidation algorithm that takes into consideration QoS metrics when planning VM migrations.

In [15] the authors propose an adaptive algorithm for dynamic voltage scaling of Web Servers subject to response time constraints. The algorithm is based on an instrumented Linux kernel which includes a QoS-aware scheduling policy, an admission control scheme and a dynamic voltage scaling scheme which minimizes the energy consumption yet ensuring that deadlines are met. In order to quickly identify the most appropriate power level, performance bounds for schedulability of aperiodic tasks are employed. With respect to [15], EASY can be implemented as a user-level application (no modification of the Operating System is needed); furthermore, EASY can deal with a distributed application, while [15] operates at the single node level.

Other DPM solutions are based on system reconfiguration steps that include powering off components while idle, and switching them to low-power operating modes when underutilized. In [16] the authors compare different solutions for reducing the power consumption of Web servers, based on DVS and batching of requests. In [17] the authors propose an autonomic management framework that uses a Markov chain quantitative model to drive the tuning of parameters of an IT system; a case study is shown involving the use of the framework for dynamic power management of disk drives. In [18] the authors describe NapSAC, a system for reducing the energy consumption of Web applications. NapSAC makes use of energy-efficient hardware and low-power sleep states to essentially “shut down” unused servers, and bring them back quickly when needed. Recent surveys [19,3,20] indicate that most of the DPM techniques developed so far are targeted at individual components (like processors or disk drives) or at specific systems (such as clusters or data centers), see for example [21,22,13].

In [23] the authors evaluate five strategies to save energy, based on various combinations of DVS and powering down/up unused servers; the authors study the performance degradation of applications with respect to the strategy used.

Several approaches consider the problem of reducing the energy consumption with SLA constraints as an optimization problem. In [4] the authors describe a dynamic optimization problem for server provisioning and DVS control for an IT infrastructure hosting multiple applications subject to response time SLAs. Three solution techniques for the optimization problem are presented: fully proactive, fully reactive and hybrid.

In [24] the authors propose a framework for self-adaptive capacity management for a virtualized hosting environment running multiple application classes. A cost model based on a two-level SLA specification is considered, where the service provider incurs a penalty when the service level requirements are violated. [25] extends the work above to handle performance/cost tradeoffs arising when operating under security attacks and energy constraints.

All papers [4,24,25] model each server (or VM, in the case of [24]) as an independent queueing center, in order to estimate the system response time at that center. This is an appropriate approximation when all application instances are independent and do not interact, but fails when the hosted application is fully distributed and spans multiple hosts. EASY uses a QN model to represent the fact that each user request may interact with more than one device before completion.

Mistral [26] is a utility-based optimization framework that optimizes a combination of performance and total power consumption for a virtualized infrastructure hosting multiple distributed applications. The system is reconfigured when variations in the monitored workload occurs. The system model used by Mistral is quite different from the one considered by EASY. The decision variables used by Mistral optimization process are essentially the number of VMs running on each host; the cost of migrating running VMs to other physical nodes is taken into consideration when deciding how to reconfigure the system.

An approach to SLA-aware adaptive resource allocation in virtualized environments is described in [27]. When a SLA violation is detected, a PUSH phase is started in which additional resources are allocated until the SLA is satisfied again; then, a PULL phase is executed in which under utilized resources are removed from the allocated pool.

The approach in [28] implements and validates a dynamic resource provisioning framework for virtualized server environments. In order to conserve power, unused machines are switched off; the cost of switching off and on physical servers is explicitly taken into account.

With respect to the above class of works, EASY deals with OLDI applications subject to response time constraints. Power management policies involving non operational ACPI states (see Section 3) require a significant overhead to reconfigure the hardware infrastructure (e.g., waiting for servers to enter/leave non operational states) and also to reconfigure the application (e.g., migrating important data away from machines being shut down). These delays are hardly tolerated by OLDI applications, for which techniques based on dynamic provisioning through dynamic allocation of VM instances are not appropriate. Therefore EASY uses operational performance states only, in order to ensure reactiveness to workload fluctuations. Note that reconfiguring the application requires specific high-level knowledge which should be embedded into the power management module; by avoiding such reconfigurations, EASY can be applied to any system.

We conclude this review by mentioning an interesting recent work [29] in which the authors make use of a new set of metrics, called *CPU gradients*, to predict the impacts of changes in processor frequency or VM capacity on the system end-to-end response time. The authors propose a framework, based on CPU gradients, aimed at minimizing the energy usage of a multi-tier environment (either virtualized or non-virtualized) subject to response time constraints. CPU gradients, as the name suggests, represents derivatives of the system response time with respect to the resource parameters (CPU frequency and VM capacity). Gradients are collected using a measurement-based approach: small perturbations in CPU frequency or VM capacity are injected into a running applications to estimate the variations of response times. CPU gradients are proposed as a measurement-based alternative to queueing network models, citing possible difficulties in building accurate QN models with little or no detailed knowledge on the application and deployment platform.

In this paper we argue that QN models can indeed be built even when no application-level knowledge is available; accurate model parameters can be collected at run-time by passive probes. Operational analysis of QN models [30] allows bounds on response time to be computed quickly [31]. We will show that EASY can represent an appropriate solution in those situations when quick system reconfiguration is important, and low invasiveness is desired.

3. The ACPI specification

Most modern CPUs support the Advanced Configuration and Power Interface (ACPI) [7], an open standard for platform-independent configuration and power management of both individual devices and entire systems. ACPI-compliant devices allow high-level components (such as the Operating System) to explicitly control the system power consumption. ACPI allows complex policies to be managed by the OS rather than a ROM BIOS; the OS can gather information at the system (hardware) and application levels in order to take better power management decisions; this would not be possible if power management features were embedded into device firmware or ROMs.

ACPI defines global system states, individual device power states and processor power states; these states, and valid transitions, are summarized in Fig. 3.

The *global states* G0–G3 are defined as follows (in order of decreasing power consumption):

G0—Working. In this state the system operates normally and application threads are executed; individual components may have their power state changed dynamically, e.g. by the user.

G1—Sleeping. In this state the system requires less power, but application threads are not being executed. The system can be brought back to working state without the need to reboot.

G2—Soft off. In this state the system power consumption is minimal. User or system software is not executed. The system must be restarted, and requires a large latency in order to go back to the working state.

G3—Mechanical Off. In this state the system is shut off through mechanical means. Except for the real-time clock, the power consumption is zero.

The *device states* D0–D3 describe the power state of individual devices attached to the system (e.g., disks drives, monitors and so on). The states are defined as follows, in order of decreasing power consumption:

D0 (Fully-On). The device is active and responsive, requiring the highest power consumption.

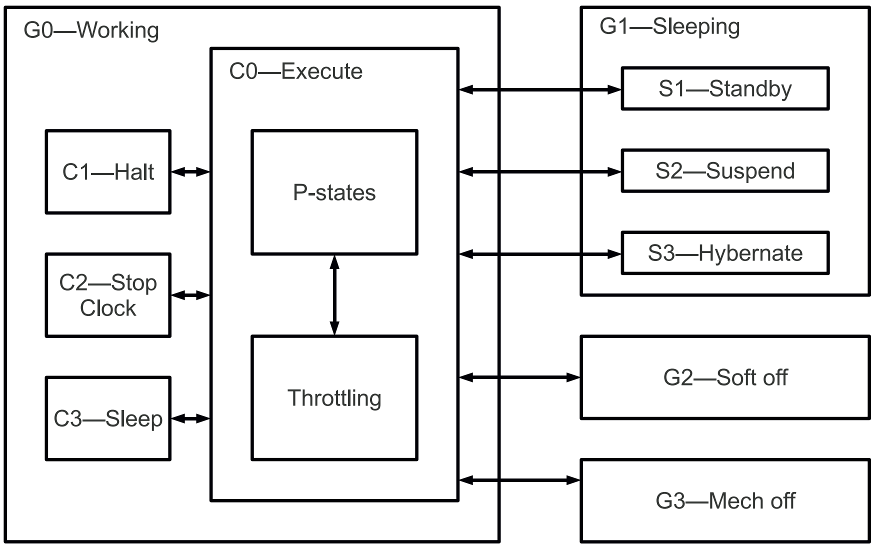

Fig. 3. ACPI states and transitions [32].

Table 1
P-states frequency, voltage and power consumption for the AMD Opteron Processor [38, Table 11, p. 14]

P-state	Frequency	Voltage	Power
P0	2.8 GHz	1.40 V	92.6 W
P1	2.6 GHz	1.35 V	90.2 W
P2	2.4 GHz	1.30 V	77.0 W
P3	2.2 GHz	1.25 V	65.7 W
P4	2.0 GHz	1.20 V	55.9 W
P5	1.8 GHz	1.15 V	47.6 W
Pmin	1.0 GHz	1.10 V	36.1 W

D1, D2, D3hot. These states require less power (D0 > D1 > D2 > D3hot), but the device is expected to preserve increasingly less context.
D3 (Off). In this state, power has been removed from the device. The device context is lost when this state is entered.

Processor power states C0–C3 are defined within the global working state G0 as follows, again in order of decreasing power consumption:

C0. The processor executes instructions normally.
C1—Halt. The processor is not executing instructions, but can return to the C0 state with minimal latency. All software-visible states are preserved.
C2—Stop-Clock. This state offers improved power savings over the C1 state, but requires a larger latency to return to an executing state. All software-visible states are preserved.
C3—Sleep. This state offers improved power savings over the C1 and C2 states. Cache coherency is not guaranteed, and must be ensured by software on resume.

Finally, the ACPI specification defines *device and processor performance states Pi* within the active states (C0 for processors and D0 for devices). Devices and processors may support up to 16 states, labeled from P0 to P16, where state *Pi* requires more energy than state $P(i+1)$, but provides better performance. Transitions between ACPI performance states incur negligible overhead; this is the reason why we focus on them in this paper.

Power reduction in operational states is achieved by dynamically adjusting the processor core voltage, clock rate or both, since the power P dissipated by a CMOS circuit satisfies the following relation [33]:

$$P \propto CV^2 f$$

where C is the capacitance, V the voltage and f the clock frequency. Various technologies are used by different vendors to achieve dynamic power reduction, such as Intel Speedstep [34], AMD PowerNow! [35], and IBM EnergyScale [36]. Additionally, AMD CoolCore [37] is capable of turning off individual parts of the processor to further reduce energy consumption and CPU temperature.

To provide some actual figures, Table 1 reports the operating voltages, frequencies and power consumption of the AMD Opteron Processor; Table 2 reports the same information for the 2 GHz VIA C7-M processor, currently offered for the mobile market.

4. Problem formulation

We now formally describe the problem addressed in this paper. We consider an application running on a distributed system made of a set $\mathcal{K} = \{1, \ldots, K\}$ of K devices; each integer $k \in \mathcal{K}$ uniquely identifies a device (e.g., a CPU, a disk, etc.). Clients interact with the application by issuing requests which are processed by the system.

Table 2
P-state frequency, voltage and power consumption for the 2 GHz VIA C7-M processor.

P-state	Frequency	Voltage	Power
P0	2.0 GHz	1.148 V	20 W
P1	1.8 GHz	1.132 V	18 W
P2	1.6 GHz	1.100 V	15 W
P3	1.4 GHz	1.052 V	13 W
P4	1.0 GHz	1.004 V	10 W
P5	800 MHz	0.844 V	7 W
P6	600 MHz	0.844 V	6 W
P7	400 MHz	0.844 V	5 W

Source: http://www.via.com.tw/en/initiatives/greencomputing/powersaver.jsp.

Processing involves requesting service from multiple devices; at the end, a response is sent back to the client.

Device k supports $L[k]$ different performance states,[2] which for notational convenience will be labeled as integers in the set $\mathcal{L}[k] = \{1, \ldots, L[k]\}$. State 1 is the state with maximum speed and higher power consumption, while state $L[k]$ offers the lowest power consumption but also the lowest performance.

We denote with $EN[k, j]$ the power consumption of device $k \in \mathcal{K}$ in state $j \in \mathcal{L}[k]$; power consumptions can be different from device to device (we allow heterogeneous systems). The total energy consumed by device k in state j over a time interval of duration T is $T \times EN[k, j]$. We require that the energy consumption rates and relative speeds are monotonically decreasing:

$$EN[k, 1] > EN[k, 2] > \cdots > EN[k, L[k]] > 0$$

We denote with $RSP[k, j]$ the *relative speed* of device k in state j, defined as:

$$RSP[k, j] = \frac{\text{Processing rate of device } \ k \ \text{ in state } \ j}{\text{Processing rate of device } \ k \ \text{ in state } \ 1} \tag{1}$$

Intuitively, $RSP[k, j]$ represents the slowdown of device k running in state j with respect to the same device operating in state 1. For example, if $RSP[k, 3] = 0.2$, then a computation requiring x time units on device k in state 1 would take $(1/0.2)x = 5x$ time units on the same device in state 3. Relative speeds can be different from device to device; we only require that processing rates are monotonically decreasing:

$$1 = RSP[k, 1] > RSP[k, 2] > \cdots > RSP[k, L[k]] > 0$$

This means that a device operating in higher-numbered performance states is slower than the same device in lower-numbered performance state.

A *system state* (or *system configuration*) is a vector $\mathbf{S} = (S[1], \ldots, S[K])$, which represents the fact that device k is in performance state $S[k] \in \mathcal{L}[k]$.

The goal of EASY is to dynamically adjust the state of each device so that the system power consumption rate is minimized, while maintaining the mean system response time below a predefined threshold $R_{\max}$. Let $\mathbf{S}(t)$ be the system configuration at time t. We want to identify a state $\mathbf{S}(t)$ which solves the following optimization problem, which we denote as $\mathcal{P}(R_{\max})$:

$$\mathcal{P}(R_{\max}) \equiv \quad \begin{array}{ll} \text{Minimize :} & E(\mathbf{S}(t)) = \displaystyle\sum_{k=1}^{K} EN[k, S[k](t)] \\ \text{Subject to :} & \begin{array}{l} R(\mathbf{S}(t)) \leq R_{\max} \\ S[k](t) \in \mathcal{L}[k] \quad \text{for all} k \in \mathcal{K} \end{array} \end{array} \tag{2}$$

[2] In this paper we make extensive use of array subscripts. In order to enhance readability of subscript indexes, we write $L[k]$ instead of L_k to denote the kth element of array L.

where $R(\mathbf{S}(t))$ is the mean system response time with configuration $\mathbf{S}(t)$. The value of the objective function $E(\mathbf{S}(t)) = \sum_{k=1}^{K} EN[k, S[k](t)]$ is the total energy consumption rate with configuration $\mathbf{S}(t)$.

It is important to observe that the response time $R(\mathbf{S}(t))$ not only depends on the system configuration $\mathbf{S}(t)$ at time t, but also on other parameters, including the workload intensity (number of concurrent users interacting with the system). Therefore, the optimization problem (2) should be solved continuously at run-time in order to allow the system to operate within the requested QoS constraints with minimum energy consumption. Of course, computing new configurations with high frequency is infeasible, as that operation would become a bottleneck. We describe in the next section a practical approach for identifying reconfiguration times and computing new system configurations.

We remark that all devices have a finite processing capacity, i.e., can process requests up to a finite maximum rate; obviously, the maximum processing capacity can be obtained when all devices operate in state 1. Nevertheless, the requests arrival rate can be larger than the maximum processing capacity, resulting in arbitrarily large delays due to queueing of requests at the bottleneck device(s). Thus, if the arrival rate of requests keeps increasing, the system response time gets larger no matter what the system configuration is. From this, we conclude that the optimization problem (2) may have no solution, e.g., when the workload intensity is higher than the maximum system capacity. In such case, EASY will return $\mathbf{S} = (1, \ldots, 1)$ as a pseudo-solution, in which all devices operate in the faster ACPI state.

Example. As a practical example, let us consider the system shown in Fig. 4; the system has $K = 4$ devices (servers) hosting a distributed application which must provide an average response time no greater than $R_{\max} = 50$ ms. Let us assume that there is a single user which submits a single request at a time, waits for its completion, and immediately submits a new request. Each request arrives at server 1, where it spends 7 ms on average; then it is routed with probability 0.5 to server 2, where it spends 35 ms, and with probability 0.5 to server 3, where it spends 30 ms on average. Finally, after additional 4 ms on server 4, the request completes execution and the result is sent back to the user.

Suppose that servers 1 and 4 do not support ACPI, while servers 2 and 3 have 3 ACPI performance states each (therefore, $L = (1, 3, 3, 1)$). The power consumptions $EN[i, j]$ of server i in state j are:

$$\begin{array}{lll} EN[1,1]=50 & & \\ EN[2,1]=50 & EN[2,2]=45 & EN[2,3]=40 \\ EN[3,1]=65 & EN[3,2]=62 & EN[3,3]=59 \\ EN[4,1]=50 & & \end{array}$$

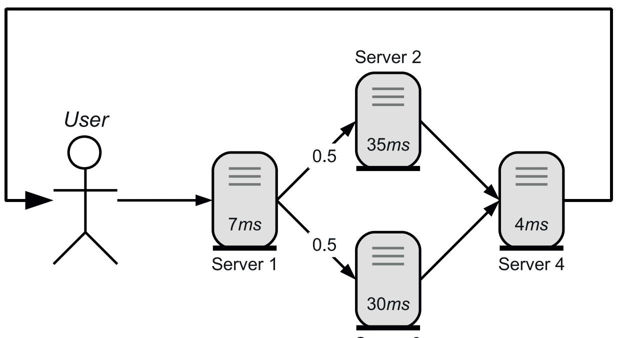

Fig. 4. Demo system.

Table 3
Estimated response time and power consumption of all valid configurations of the system in Fig. 4. Underlined configurations have estimated mean response time not exceeding 50 ms.

Configuration $\mathbf{S}$	Resp. time $R(\mathbf{S})$	Power $E(\mathbf{S})$
$\underline{(1,1,1,1)}$	43.000	215.00
$\underline{(1,1,2,1)}$	46.176	212.00
$\underline{(1,1,3,1)}$	49.000	209.00
$\underline{(1,2,1,1)}$	46.500	210.00
$\underline{(1,2,2,1)}$	49.676	207.00
(1, 2, 3, 1)	52.500	204.00
$\underline{(1,3,1,1)}$	49.000	205.00
(1, 3, 2, 1)	52.176	202.00
(1, 3, 3, 1)	55.000	199.00

and the relative speeds $RSP[i, j]$ are:

$$\begin{array}{lll} RSP[1,1]=1.00 & & \\ RSP[2,1]=1.00 & RSP[2,2]=0.80 & RSP[2,3]=0.70 \\ RSP[3,1]=1.00 & RSP[3,2]=0.85 & RSP[3,3]=0.75 \\ RSP[4,1]=1.00 & & \end{array}$$

With the parameters above, the mean system response time $R(\mathbf{S})$ with configuration $\mathbf{S} = (S[1], S[2], S[3], S[4])$ can be estimated as:

$$R(\mathbf{S}) = \frac{7\text{ ms}}{RSP[1, S[1]]} + \frac{0.5 \times 35\text{ ms}}{RSP[2, S[2]]} + \frac{0.5 \times 30\text{ ms}}{RSP[3, S[3]]} + \frac{4\text{ ms}}{RSP[4, S[4]]} \quad (3)$$

Table 3 shows the estimated mean response time $R(\mathbf{S})$ and power consumption $E(\mathbf{S})$ for each configuration $\mathbf{S}$. We underline the configurations whose system response time does not exceed the threshold $R_{\max} = 50$ ms; among these, the one with lower power consumption is (1, 3, 1, 1): this would be the desired result of the optimization problem (2).

In general the scenario can be much more complex than the one shown in this simple example. First of all, multiple users can be submitting requests at the same time; hence, requests can queue at the various servers interfering with each other, so that Eq. (3) can no longer be used to estimate the system response time. Additionally, some of the parameters (service times, routing probabilities) can be difficult if not impossible to estimate in advance, and can also fluctuate over time. Finally, the number of possible configurations can grow exponentially as the number of devices increases, making the analysis of all configurations not possible. We will address all these issues in the rest of this paper.

5. EASY architecture

The optimization problem (2) is parametrized by the continuous variable t (time). In order to make the problem tractable, we partition the time into non overlapping, contiguous intervals of duration Δt. With a slight abuse of notation we denote with $t_1, t_2, \ldots$ both the end of the intervals, and the intervals themselves.

EASY is a reactive system based on the Monitor-Analyze-Plan-Execute control loop shown in Fig. 5. During the *Monitor* step, EASY measures the response time, throughput and individual device utilizations by collecting samples on the running system. At the end of each observation period, this information is used in the *Analyze* step to instantiate a performance model based on Queueing Networks. The model is used in the *Plan* step to quickly analyze different

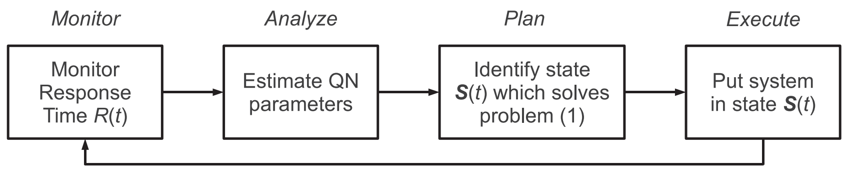

Fig. 5. Monitor-Analyze-Plan-Execute control loop.

configurations, until an approximate solution of the optimization problem (2) is found. The newly found system configuration is finally applied during the *Execute* step.

Fig. 6 shows a generic server farm enhanced with additional components (shown as gray boxes) required by EASY. We use the standard UML notation for deployment diagrams [39], where three-dimensional boxes represent physical devices, while rectangles represent software components.

The frontend is the access point to the infrastructure, and is usually responsible for admission control and load balancing. The frontend is enhanced with two additional components, the *monitor* and the *controller*. The monitor is a passive observer which simply forwards all requests to the system; while it does so, it measures the mean system response time $\overline{R}(t)$, the system throughput $\overline{X}(t)$ and individual device utilizations $\overline{U}[k](t)$, under the current configuration $\mathbf{S}(t)$ during the interval t (details are given in Section 6). The system response time and throughput are measured by passive analysis of user requests within the frontend; utilizations of ACPI-capable devices are collected by software *probes* residing on each host of the server farm. The get() interface of the probes is used to read the current ACPI level and utilization of devices within the same host (e.g., multiple CPUs, disks, network interfaces and so on). The *controller* is activated at the end of each observation interval, and uses the parameters collected by the monitor to identify a new state which solves the optimization problem (2). The new state is sent to the probes through their set() interface; each probe triggers the appropriate ACPI state changes to devices it controls.

At the end of each observation period, if the measured system response time $\overline{R}(t)$ is below the threshold $R_{\max}$, EASY tries to slow down some devices by setting them to higher numbered performance states; on the other hand, if the response time is above $R_{\max}$, EASY speeds up bottleneck devices by setting them to lower numbered performance states. To prevent excessive reconfigurations when $\overline{R}(t)$ jumps above and below the threshold $R_{\max}$, we use an *hysteresis* mechanism implemented through two additional thresholds R_{low} and R_{high}, such that $R_{\text{low}} < R_{\text{high}} < R_{\max}$. A reconfiguration is triggered only when $\overline{R}(t) > R_{\text{high}}$ or $\overline{R}(t) < R_{\text{low}}$.

Algorithm 1 *(The EASY algorithm).*

```
Require: Thresholds $R_{low} < R_{high} < R_{max}$
1:  Let S be the initial system configuration
2:  loop
3:    $t$ := current time
4:    Measure $\overline{U}(t), \overline{X}(t), \overline{R}(t)$ over an interval of duration $\Delta t$
5:    if ($\overline{R}(t) > R_{high}$) then
6:      $\mathbf{S}'$ := Speedup($\mathbf{S}, \overline{U}(t), \overline{X}(t), \overline{R}(t)$)
7:    els if $\left(\overline{R}(t) < R_{low}\right)$ then
8:      $\mathbf{S}'$ := Slowdown($\mathbf{S}, \overline{U}(t), \overline{X}(t), \overline{R}(t)$)
9:    Apply system state $\mathbf{S}'$
10:   $\mathbf{S}$ := $\mathbf{S}'$
```

Algorithm 1 shows the EASY control loop. At each iteration, the system response time $\overline{R}(t)$, throughput $\overline{X}(t)$ and device utilizations $\overline{U}[k](t)$ are measured during a time interval of duration Δt (line 4). If the response time $\overline{R}(t)$ is larger than R_{high}, EASY identifies a new configuration by switching bottleneck devices to better performing states (line 6). If $\overline{R}(t)$ is lower than R_{low}, EASY tries to switch devices to power states which require less energy (line 8). The procedures SPEEDUP() and SLOWDOWN() solve the optimization problems $\mathcal{P}(R_{\text{low}})$ and $\mathcal{P}(R_{\text{high}})$, respectively: they identify a new system configuration $\mathbf{S}'$ such that the estimated response time is less than R_{high}.

By choosing appropriate values for R_{low}, R_{high} and for the monitoring interval Δt, it is possible to achieve different trade-offs between reactiveness and sensitivity to changes in the system response time. Specifically, the duration Δt of the monitoring interval controls the reactiveness of EASY: lower values result in faster reaction times, but also increase the overhead due to the computation of new configurations. The parameters R_{low} and R_{high} control how EASY is sensitive to small deviations of the average response time $\overline{R}(t)$ from the threshold $R_{\max}$.

6. System modeling

In order to estimate the system response time under different configurations, EASY makes use of a suitable performance model. We model the system as a single-class, closed QN. The QN model

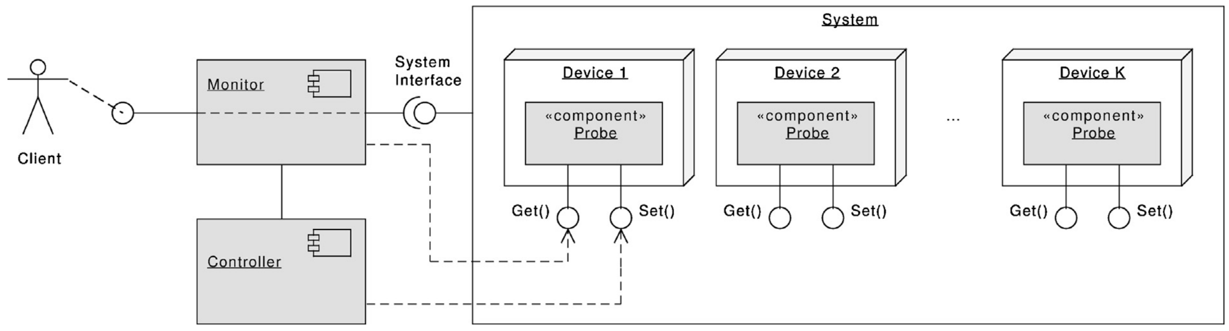

Fig. 6. System architecture; components used by EASY are shown in gray.

used by EASY contains K service centers, each one representing an individual ACPI-capable device. Requests circulate through the system, joining the queues associated to devices they get service from. Once service is complete, a request can be forwarded to another device, or may be sent back to the user who submitted it.

In a closed network there is a fixed number of requests which circulate through the system. This represents the fact that each client can have some fixed, maximum number of outstanding requests, and will wait for one of those requests to complete before submitting another one. We allow the number of clients to vary over time, since every Δt time units a new network is built and analyzed from scratch.

As already observed, efficiency of the reconfiguration algorithm is of primary importance for EASY, as it must operate on-line and should compute new system configurations as fast as possible. A special class of QNs, called product-form QNs, have a simple closed form expression of the stationary state distribution, so that average performance measures can be computed efficiently [40] (in order to make this paper self-contained, we included additional details on QNs in Appendix B).

While product-form QNs must satisfy some constraints which may reduce their accuracy in modeling real-world systems (see [40] for details), we argue that they are still accurate enough for guiding the process of identifying new configurations. The accuracy of a performance model is both related to its expressiveness (how precisely the model captures the actual system behavior), and to the accuracy of the model input parameters. In EASY, model parameters come from the monitor, which is not always up-to-date with the running system. In this situation there is very little or no gain in using more sophisticated (but possibly slower to analyze) models.

The solution of the QN model requires numerical parameters which can be derived from the system throughput, response time, and individual device utilizations. These parameters can be estimated by collecting samples over each observation period of duration Δt. Specifically, let C be the number of user requests that the system processed within an observation interval, and $T_1, T_2, \ldots, T_C$ are the times taken to process the C requests, then the system response time is estimated as $\sum_{c=1}^{C} T_c/C$, and the throughput as $C/\Delta t$. The device k utilization is the fraction of the interval Δt in which device k was busy (i.e., processing some request); by definition, $\overline{U}[k](t) \in [0, 1]$.

We define the service demand $D[k, S[k](t)]$ of device k in state $S[k](t)$ as the total time spent by a request in device k. Applying the Utilization Law [41], we have:

$$D[k, S[k](t)] = \frac{\overline{U}[k](t)}{\overline{X}(t)} \tag{4}$$

The service demand $D[k, l]$ for a generic state $l \in \mathcal{L}[k]$ can be estimated as:

$$D[k, l] = \frac{\text{Processing rate of device } k \text{ in state } l}{\text{Processing rate of device } k \text{ in state } S[k](t)} \times \frac{\overline{U}[k](t)}{\overline{X}(t)}$$

which, applying Eq. (1), can be rewritten as:

$$D[k, l] = \frac{RSP[k, S[k](t)]}{RSP[k, l]} \times \frac{\overline{U}[k](t)}{\overline{X}(t)} \tag{5}$$

The number of requests in the system $N(t)$ at time t can be computed from the throughput $\overline{X}(t)$ and response time $\overline{R}(t)$ using Little's law [42]:

$$N(t) = \overline{X}(t) \times \overline{R}(t) \tag{6}$$

Algorithm 2 (. *EstimateRespT* $(\mathbf{S}', \mathbf{S}, \overline{U}, \overline{X}, \overline{R}) \to R(\mathbf{S}')$).

```
Require: $\mathbf{S}'$ arbitrary state
Require: $\mathbf{S} = \mathbf{S}(t)$ current state
Require: $\overline{U} = \overline{U}(t)$ current device utilizations
Require: $\overline{X} = \overline{X}(t)$ current system throughput
Require: $\overline{R} = \overline{R}(t)$ current system response time
Ensure: $R(\mathbf{S}')$ estimated system response time in state $\mathbf{S}'$
  $N := \overline{X} \times \overline{R}$                                   {Little's Law (6)}
  for all $k \in \mathcal{K}$ do
    $D[k] := D[k, S'[k]] := \frac{RSP[k,S[k]]}{RSP[k,S'[k]]} \times \frac{\overline{U}[k]}{\overline{X}}$    {Service demands (5)}
  $D_{\max} := \max\{D[k] \mid k \in \mathcal{K}\}$
  $D_{\mathrm{tot}} := \sum_{k \in \mathcal{K}} D[k]$
  $D_{\mathrm{ave}} := D_{\mathrm{tot}}/K$
  $R^- := \max\{ND_{\max}, D_{\mathrm{tot}} + (N-1)D_{\mathrm{ave}}\}$        {Lower bound}
  $R^+ := D_{\mathrm{tot}} + (N-1)D_{\max}$                          {Upper bound}
  Return $(R^+ + R^-)/2$
```

Given the current configuration $\mathbf{S} = \mathbf{S}(t)$, measured device utilizations $\overline{U} = \overline{U}(t)$, system throughput $\overline{X} = \overline{X}(t)$ and response time $\overline{R} = \overline{R}(t)$, function ESTIMATERESPT() shown in Algorithm 2 estimates the system response time $R(\mathbf{S}')$ for any configuration $\mathbf{S}'$. The estimate is obtained as the average of the upper and lower response time bounds (R^+ and R^-, respectively) computed using Balanced System Bounds (BSB) [31].

While the result returned by function ESTIMATERESPT() is only an estimate, the exact steady-state response time could be computed using the Mean Value Analysis (MVA) algorithm [43]. However, MVA requires $O(NK)$ operations for analyzing a QN with K queueing centers and N requests, while Algorithm 2 only requires $O(K)$ operations, which does not depend on N. We show in Section 9 that, despite this approximation, EASY provides almost optimal configurations.

7. Solving the optimization problem using Mixed Integer Programming

We now describe a formulation of the optimization problem $\mathcal{P}(R_{\max})$ as a MIP problem which we call $\mathcal{P}_{\mathrm{MIP}}(R_{\max})$. While Integer Linear Programming is NP-complete, existing MIP solvers can handle medium-sized problems quite readily. Therefore, we use $\mathcal{P}_{\mathrm{MIP}}(R_{\max})$ as a benchmark against which a faster heuristic, described in Section 8, will be compared.

Let $D[k, l]$ be the service demand of device k when operating in performance state $l \in \mathcal{L}[k]$, as defined in (5). We encode the solution of the optimization problem (2) in a binary matrix $X[k, l]$, such that $X[k, l] = 1$ if and only if device k must be set in state l to get the minimum energy consumption rate. As described in Section 6, we estimate the system response time $R(\mathbf{S}')$ for an arbitrary configuration $\mathbf{S}'$ using Balanced System Bounds. Then, problem $\mathcal{P}(R_{\max})$ can be formulated as shown in Fig. 7.

It should be observed that the problem $\mathcal{P}_{\mathrm{MIP}}(R_{\max})$ is not linear, due to constraints (*) on R^- and (**) on $D_{\max}$. Fortunately, we can obtain an equivalent linear problem by replacing (*) with:

$$\begin{aligned} R^- &\geq ND_{\max} \\ R^- &\geq D_{\mathrm{tot}} + (N-1)D_{\mathrm{ave}} \end{aligned}$$

and replacing (**) with the following K constraints:

$\mathsf{P}_{\text{MIP}}(R_{\max}) \equiv$

Given:	$\mathcal{K} = \{1, \ldots, K\}$,	Set of devices
	$\mathcal{L}[k] = \{1, \ldots, L[k]\}$,	$k \in \mathcal{K}$; Set of performance states of device k
	$D[k, l]$,	$k \in \mathcal{K}$, $l \in \mathcal{L}[k]$: estimated service demand of device k in performance state l, computed using Eq. (5)
	$EN[k, l]$,	$k \in \mathcal{K}$, $l \in \mathcal{L}[k]$: energy consumption rate of device k in performance state l
	$R_{\max}$,	Maximum system response time
	N	Number of users currently connected to the system
Define:	$X[k, l]$,	$k \in \mathcal{K}$, $l \in \mathcal{L}[k]$: $X[k, l] = 1$ if and only if device k should be set in performance state l, 0 otherwise
Minimize:	$\sum_{k \in \mathcal{K}} \sum_{l \in \mathcal{L}[k]} X[k, l] \times EN[k, l]$,	Total energy consumption
Subject to:	$R \leq R_{\max}$,	The system response time should not exceed $R_{\max}$
	$\sum_{l \in \mathcal{L}[k]} X[k, l] = 1$,	$k \in \mathcal{K}$: Each device must be in one performance state only
	$R = (R^+ + R^-)/2$,	The system response time is estimated as the average of upper and lower bounds
(*)	$R^- = \max\left(N D_{\max}, D_{\text{tot}} + (N-1) D_{\text{ave}}\right)$,	Lower bound of the response time
	$R^+ = D_{\text{tot}} + (N-1) D_{\max}$,	Upper bound of the response time
	$D_{\text{tot}} = \sum_{k \in \mathcal{K}} D[k]$,	
(**)	$D_{\max} = \max_{k \in \mathcal{K}} D[k]$,	
	$D_{\text{ave}} = D_{\text{tot}} / K$,	
	$D[k] = \sum_{l \in \mathcal{L}[k]} X[k, l] \times D[k, l]$,	$k \in \mathcal{K}$: $D[k]$ is the service demand of device k with the current configuration encoded by matrix $X[k, l]$

Fig. 7. Mixed Integer Programming problem $\mathcal{P}_{\text{MIP}}(R_{\max})$.

$$D_{\max} \geq D[k] \quad \textit{forall} \quad k \in \mathcal{K}$$

The resulting problem is linear, and can be solved using many existing software packages (e.g., GLPK,[3] CPLEX,[4] `lp_solve`,[5] just to name a few). Solving MIP problems can be computationally prohibitive [44]; however, MIP solvers allow the user to specify a maximum computation time; if this time elapses before the optimal solution is found, the "best" feasible solution identified so far is returned.

8. Heuristic solution of the optimization problem

In this section we describe a heuristic solution algorithm for solving the optimization problem (2). The heuristic is based on the following idea: given the current configuration **S** we iteratively compute the new configuration **S**′ by switching one device at a time to a higher or lower numbered performance state.

Specifically, if the current response time $R(t)$ is above the threshold R_{high} ($R(t) > R_{\text{high}}$), we identify the bottleneck device B and switch it to a lower numbered ACPI state, such that the device operates faster. This process is iterated until either (i) the estimated system response time becomes lower than R_{high}, or (ii) all devices are in state 1, and thus cannot be made any faster.

On the other hand, if the current response time $R(t)$ is below the threshold R_{low} ($R(t) < R_{\text{low}}$), we identify the device with lower utilization, and switch it to a higher numbered ACPI state. The process stops when no device can be slowed down without making the estimated system response time higher than R_{high}.

The QN performance model plays a fundamental role in the reconfiguration process. As will be described in Sections 8.1 and 8.2 below, the QN is used to estimate the utilization of devices and the system response time for the new configurations, without interfering with the real system. In Section 8.3 we analyze the computational costs of the heuristic, and characterize the results it provides.

8.1. Speeding up the system

When $R(t) > R_{\text{high}}$, the response time can be improved by identifying and removing the system bottleneck. From

[3] http://www.gnu.org/software/glpk/.
[4] http://www.ampl.com/.
[5] http://lpsolve.sourceforge.net/.

queueing theory, we know that the bottleneck is the device with highest service demand [41]. Since our goal is to minimize the power consumption, we take into consideration also the current ACPI state to give preference to devices which have high service demand but low energy consumption. Specifically, if $\mathbf{S}'$ is the configuration being considered, we define the bottleneck k as the device for which the ratio $D[k, S'[k]]/EN[k, S'[k]]$ is maximized.

The details are shown in Algorithm 3. The procedure SPEEDUP($\mathbf{S}$, $\mathbf{U}$, X, R) takes as input the current configuration $\mathbf{S}$, the measured device utilizations $\mathbf{U}$, and the system throughput X and response time $R > R_{\text{high}}$. The procedure computes a new configuration $\mathbf{S}'$ for which the estimated response time is less than R_{high}.

$\mathbf{S}'$ is computed iteratively, by switching one device at a time to a faster ACPI performance state. At each iteration we compute the set $\mathcal{C}$ of devices which are in a performance state strictly greater than one; therefore, $\mathcal{C}$ is the set of devices which can be made faster. The bottleneck is the device $B \in \mathcal{C}$ for which the ratio $D[B, S'[B]]/EN[B, S'[B]]$ is maximum (line 5), $S'[B]$ being the performance state of B in configuration $\mathbf{S}'$. Then, device B is switched to ACPI state $S'[B] - 1$.

The procedure SPEEDUP stops either when (i) the estimated system response time becomes less than R_{max}, or (ii) all devices have been set to state 1 (thus $\mathcal{C}$ becomes empty).

Algorithm 3 *(SPEEDUP($\mathbf{S}$, $\mathbf{U}$, X, R) → $\mathbf{S}'$).*

```
Require: S current system state
Require: U current device utilizations
Require: X current system throughput
Require: R current system response time, $R > R_{\text{high}}$
Ensure: S′ new system state
1:  Compute $D[k, j]$ using (5), for all $k \in \mathcal{K}, j \in \mathcal{L}[k]$
2:  $\mathbf{S}' := \mathbf{S}$
3:  $\mathcal{C} := \{k \in \mathcal{K} \mid S'[k] > 1\}$                    {Candidate set}
4:  while ($\mathcal{C} \neq \emptyset$) do
5:      $B := \text{argmax}_k \left\{ \frac{D[k,S'[k]]}{EN[k,S'[k]]} \middle| k \in \mathcal{C} \right\}$
6:      $S'[B] := S'[B] - 1$                                 {Speed up device B}
7:      f$R_{\text{est}}$ := EstimateRespT($\mathbf{S}'$, $\mathbf{S}$, $\mathbf{U}$, $X$, $R$)
8:      if ($R_{\text{est}} < R_{\text{high}}$)
9:          Break                                          {Complete}
10:     $\mathcal{C} := \{k \in \mathcal{K} | S'[k] > 1\}$          {Recompute the candidate set}
11: Return S′
```

8.2. Slowing down the system

Algorithm 4 is used to reduce the total energy consumption rate by selectively slowing down non-bottleneck devices. Again, we use a greedy approach in which a new system configuration $\mathbf{S}'$ is derived from the current one, by slowing down one device U at a time. U is selected among a set $\mathcal{C}$ of candidates, which initially contains the index of all devices which are not in their highest numbered (slower) performance state.

The device U satisfies the following two properties:

- It is the device whose slowdown produces the minimum increase in the system response time with maximum reduction of energy consumption rate (line 5);
- After setting U in state $S'[U] + 1$, the estimated system response time is below the threshold R_{high} (lines 6 and 7).

Algorithm 4 *(SLOWDOWN($\mathbf{S}$, $\mathbf{U}$, X, R) → $\mathbf{S}'$).*

```
Require: S current system state
Require: U[k] current utilization of device k
Require: X current system throughput
Require: R current system response time, $R < R_{\text{low}}$
Ensure: S′ new system state
1: Compute $D[k, j]$ using (5), for all $k \in \mathcal{K}, j \in \mathcal{L}[k]$
2: $\mathbf{S}' := \mathbf{S}$
3: $\mathcal{C} := \{k \in \mathcal{K} | S'[k] < L[k]\}$          {Devices which can be made slower}
4: while ($\mathcal{C} \neq \emptyset$) do
5:     $U := \text{argmin}_k \left\{ \frac{D[k,S'[k]+1]}{EN[k,S'[k]+1]} \middle| k \in C \right\}$
6:     $S'[U] := S'[U] + 1$                        {Try to slow down device U}
7:     $R_{\text{est}}$ := EstimateRespT($\mathbf{S}'$, $\mathbf{S}$, $\mathbf{U}$, $X$, $R$)
8:     if ($R_{\text{est}} > R_{\text{max}}$) then
9:         $S'[U] := S'[U] - 1$                    {Rollback configuration for device U}
10:        $C := \mathcal{C} \setminus \{U\}$        {Device U will no longer be considered}
11:    els if ($S'[U] = L[U] - 1$) then
12:        $C := \mathcal{C} \setminus \{U\}$
13: Return S′
```

Device U is removed from $\mathcal{C}$ if it can not be made any slower (line 11), or if switching it to a slower performance state violates the constraint $R > R_{\text{high}}$. The procedure terminates when the set $\mathcal{C}$, computed at the beginning of each iteration, is empty.

8.3. Efficiency and optimality considerations

Algorithms 3 and 4 execute a number of iterations in which a single device is reconfigured at a time; in particular, at each iteration one device k is switched from state $S[k]$ to $S[k] + 1$ (in the SLOWDOWN procedure), or from state $S[k]$ to $S[k] - 1$ (in the SPEEDUP procedure).

In order to analyze the solution $\mathbf{S}'$ computed by the procedures above, we introduce some notation. Let $\mathbf{P}$ and $\mathbf{Q}$ two arbitrary system configurations. We say that $\mathbf{P} \prec \mathbf{Q}$ if, for all $k \in \mathcal{K}$, $P[k] \leq Q[k]$, the inequality being strict for at least one value of k.

Using the assumption that power consumption rates are monotonically decreasing (see Section 4), we conclude that $\mathbf{P} \prec \mathbf{Q}$ implies $E(\mathbf{P}) < E(\mathbf{Q})$. Also, since switching a device k to a higher performance state reduces its service demand, the following Lemma holds:

Lemma 1. *If$\mathbf{P} \prec \mathbf{Q}$, then$R(\mathbf{P}) > R(\mathbf{Q})$, where$R(\mathbf{S})$is the estimated system response time computed by analyzing the QN model.*

Lemma 1 derives from known monotonicity properties of QN models [45], and it basically states that if we switch a single device to a upper (lower) performance level, the expected system response time increases (decreases). Note that the Lemma is true also if $R(\mathbf{S})$ is the exact response time of the QN model as computed by the MVA algorithm.

Using the properties above, we can characterize the solutions computed by Algorithms 3 and 4. The invocation of SPEEDUP($\mathbf{S}$, $\mathbf{U}$, X, R) returns a new configuration $\mathbf{S}'$ such that $\mathbf{S}' \prec \mathbf{S}$. From Lemma 1 we have that $R(\mathbf{S}') < R(\mathbf{S})$. Moreover, by inspecting the code we can observe that there exists no configuration $\mathbf{S}'' \succ \mathbf{S}'$ such that $R(\mathbf{S}') < R(\mathbf{S}'') \leq R_{\text{high}}$. Hence, $\mathbf{S}'$ is *Pareto optimal* for problem $\mathcal{P}(R_{\text{high}})$. Similarly, the configuration $\mathbf{S}'$ returned by procedure SLOWDOWN is Pareto optimal for problem $\mathcal{P}(R_{\text{high}})$, in the sense that there exists no configuration $\mathbf{S}'' \succ \mathbf{S}'$ such that $R(\mathbf{S}') < R(\mathbf{S}'') \leq R_{\text{high}}$.

The execution time of Algorithms 3 and 4 is the product of the number of iterations $O\left(\sum_{k \in \mathcal{K}} |S'[k] - S[k]|\right)$, and the cost $O(K)$ of each iteration. Hence, the total cost of procedure SPEEDUP and SLOWDOWN is $O\left(K \times \sum_{k \in \mathcal{K}} |S'[k] - S[k]|\right)$, which in the worst case is $O\left(K \times \sum_{k \in \mathcal{K}} L[k]\right)$. Note that the total number of possible configurations is $O\left(\prod_{k\in\mathcal{K}} L[k]\right)$, hence the complete exploration of the solution space would be infeasible even for small systems.

9. Numerical evaluation

In this section we assess the effectiveness of EASY by means of a set of synthetic test cases which have been evaluated numerically. We consider a system with K devices. In order to limit the number of simulation parameters, we assume that all devices support the same number L of ACPI performance states. We consider all combinations of $K \in \{10, 20, 30, 50\}$ and $L \in \{2, 4, 6\}$. The relative speed $RSP[k, j]$ of device k in state j is defined as a linear function of j, with minimum value $RSP[k, L] = 0.3$ and maximum $RSP[k, 1] = 1$. Therefore

$$RSP[k, j] = 1.0 - \frac{j-1}{L-1} \times 0.7$$

The energy consumption rate $EN[k, j]$ is normalized in the range [0.4, 1], and is defined as:

$$EN[k, j] = 1.0 - \frac{j-1}{L-1} \times 0.6$$

Linear dependency of the energy consumption rate and relative speed is consistent with data from actual processors (see Tables 1 and 2).

We performed a time-stepped simulation with $T = 300$ steps. Algorithm 1 is executed every $\Delta t = 5$ time steps. We simulated a variable workload by changing the number of users $N(t)$ at step t. In order to use realistic data, we monitored a real large scale system by collecting the total number of online players connected to the RuneScape MMOG[6] in May 2011. RuneScape is a Fantasy MMOG where players can travel across the fictional medieval realm. Players use an ordinary Web browser to connect to one of the available RuneScape servers; the servers are located in different world regions, so that communication latency can be minimized. During peak hours, more than 200,000 players are connected to the system (the load is split across the regional servers); this number becomes as low as about 110,000 players during off-peak hours. We extracted a subset of the data spanning 4 days; the data have been down-sampled in order to fit four days in $T = 300$ time steps. Finally, the sampled data have been rescaled so that the maximum number of users is about 100 (a workload which can be reasonably handled by the system for all values of K and L).

The service demand $D[k](t)$ for device k at simulation step t has been generated by drawing uniformly distributed random values in the range [0.2, 1.0]; this ensured that service demands are not constant over time. We smoothed the service demand data by computing moving averages for each device, using a window size of 10 steps.

For each experiment the "hard" threshold $R_{\max}$ is defined as follows. Let $t_{\max} = \text{argmax}\{N(t) | t = 1, \ldots, T\}$ be the time step at which there is the maximum number of concurrent users. We use the MVA algorithm to compute the system response time $\overline{R}(t_{\max})$ at step $t_{\max}$, using the number of requests $N(t_{\max})$ as above, and with all devices set in performance state L (slower). We then define $R_{\max} = 0.8 \times \overline{R}(t_{\max})$, $R_{\text{high}} = 0.9 \times R_{\max}$ and $R_{\text{low}} = 0.8 \times R_{\max}$. The above definition of $R_{\max}$ ensures that, at each time step $t \in \{1, \ldots, T\}$, there always exists a configuration $\mathbf{S}(t)$ which solves the optimization problem $\mathcal{P}(R_{\max})$.

EASY is executed on-line, that is it finds a new configuration at time step t by considering only the configuration at the previous step $t-1$ and the number of currently active requests N_t. The observed system response time $\hat{R}(t)$ at time t is computed as follows: first, we compute the system response time from the QN model using MVA; then, we multiply this value by a random number uniformly distributed in [0.9, 1.1]. This is used to simulate the fact that, in practice, the QN model will not produce the correct response time estimates.

We implemented Algorithms 3 and 4 in GNU Octave [46], an interpreted language for numerical computations. We implemented the main control loop (Algorithm 1) and procedures SPEEDUP() and SLOWDOWN() (Algorithms 3 and 4). We compared the results with the exact solutions of the MIP problem $\mathcal{P}_{\text{MIP}}(R_{\text{high}})$ computed by GLPK with a timeout of 60 seconds. While in most cases the optimal solution was found before the timeout expired, GLPK is significantly slower than the heuristic (see Fig. 12, described below). All tests have been performed on an AMD Athlon 64 X2 3800+dual core processor with 4 GB of RAM running Linux 2.6.32; we used GNU Octave version 3.2.3 and GLPK version 4.45.

For each combination of K and L, we performed 10 independent simulation runs, using different seeds for generating independent sequences of pseudo-random numbers. We then computed average performance measures and 90% confidence intervals.

Fig. 8 shows an example of the results for a system with $K = 10$ components operating at $L = 2$ quality levels. Fig. 8(a) shows a run where EASY uses the heuristic described in Section 8, while Fig. 8(b) shows a run where we compute the exact solution of the MIP problems with the help of GLPK. The top part of the plot shows the observed system response time $\overline{R}(t)$ at step t (thick line), together with the moving average (thin line); the latter is used by EASY to decide when a reconfiguration should be triggered, according to Algorithm 1. Reconfiguration points are shown as small circles: these are the times when either procedure SPEEDUP() or SLOWDOWN() have been invoked. The middle part of Fig. 8 shows the power consumption $E(\mathbf{S}(t))$; horizontal lines show the minimum energy consumption rate, corresponding to the configuration in which all devices are in performance state L, and the maximum energy consumption rate, corresponding to the configuration in which all devices are in the fastest performance state 1. Finally, the bottom part of Fig. 8 shows the number of users $N(t)$ at time step t.

We now describe the performance metrics which have been considered in our study, and discuss the results obtained from the simulation runs. For completeness, the raw data are reported in Tables 4 and 5. Table 6 allows quick comparison between the heuristic algorithm and the MIP solver; for each row, we underline the best result whose confidence interval does not overlap with the alternative.

9.1. Number of SLA violations

This is the number of simulation steps in which the response time constraints $\overline{R}(t) \le R_{\max}$ has been violated. This is a *lower is better* metric, which means that lower values are preferred.

Fig. 9 shows the results obtained with the heuristic and the MIP solver. We observe that, in most cases, the number of violations produced by the heuristic and by the MIP solver are comparable at this confidence level; this means that the configurations produced by finding an optimal solution of the MIP problem are not significantly better than those produced by the heuristic. However, if we look at the average values alone (height of histograms), we observe that the heuristic produces marginally less SLA violations than the MIP solver. This may seem counter intuitive, as solutions of the MIP problem are expected to be the "best" possible configurations. The observed results can be explained by considering that the configurations produced by the heuristic are in general sub-optimal, which

[6] http://www.runescape.com/.

Table 4
Results for the heuristic approach.

K	L	SLA violations	ΔR	E_{tot}	E_{ovr}	Exec. time
10	2	8.90 ± 2.21	76.29 ± 28.12	1454.70 ± 18.88	0.14	4.43 ± 0.03
10	4	8.60 ± 1.03	71.42 ± 16.65	1292.90 ± 3.10	0.05	4.73 ± 0.03
10	6	8.90 ± 2.45	67.75 ± 24.58	1256.88 ± 3.47	0.03	5.03 ± 0.03
20	2	16.70 ± 2.49	137.15 ± 26.16	3569.40 ± 39.91	0.32	5.35 ± 0.04
20	4	17.00 ± 1.83	140.08 ± 24.86	2822.20 ± 11.28	0.12	5.92 ± 0.06
20	6	14.30 ± 2.24	116.35 ± 26.47	2667.84 ± 9.26	0.07	6.62 ± 0.07
30	2	25.10 ± 2.40	267.09 ± 48.80	5627.70 ± 27.37	0.38	6.30 ± 0.12
30	4	16.80 ± 1.95	172.47 ± 33.57	4386.00 ± 9.39	0.15	7.52 ± 0.15
30	6	17.60 ± 2.12	159.38 ± 30.82	4124.64 ± 5.11	0.10	8.27 ± 0.17
50	2	26.60 ± 1.97	246.11 ± 31.73	9517.80 ± 51.00	0.39	7.61 ± 0.11
50	4	16.30 ± 1.50	140.97 ± 27.14	7484.70 ± 10.14	0.16	9.98 ± 0.18
50	6	15.20 ± 1.54	182.25 ± 43.83	7064.34 ± 7.46	0.12	12.12 ± 0.25

Table 5
Results for the approach involving exact solution of the MIP problem.

K	L	SLA violations	ΔR	E_{tot}	E_{ovr}	Exec. time
10	2	9.70 ± 1.59	73.67 ± 14.63	1418.70 ± 13.87	0.12	4.50 ± 0.04
10	4	9.50 ± 2.01	79.50 ± 28.47	1277.20 ± 2.76	0.04	4.64 ± 0.04
10	6	9.70 ± 1.37	64.31 ± 15.39	1246.68 ± 1.89	0.03	4.81 ± 0.04
20	2	16.50 ± 1.67	115.84 ± 14.10	3289.80 ± 31.09	0.25	4.80 ± 0.06
20	4	17.60 ± 2.32	131.35 ± 24.20	2740.80 ± 13.71	0.09	5.90 ± 0.09
20	6	16.20 ± 1.54	145.47 ± 21.15	2627.82 ± 5.75	0.06	8.32 ± 0.36
30	2	29.30 ± 3.10	244.69 ± 19.78	5218.20 ± 59.86	0.30	12.01 ± 0.21
30	4	20.30 ± 1.52	173.08 ± 39.12	4278.10 ± 10.03	0.13	54.86 ± 15.98
30	6	16.30 ± 1.57	186.43 ± 29.05	4089.18 ± 11.03	0.09	40.47 ± 9.65
50	2	33.30 ± 1.91	310.63 ± 41.28	8827.80 ± 38.44	0.31	195.58 ± 19.93
50	4	17.90 ± 2.45	208.40 ± 48.19	7336.50 ± 17.71	0.15	339.11 ± 31.76
50	6	15.80 ± 2.04	191.60 ± 59.64	6998.70 ± 7.18	0.11	146.01 ± 36.48

means that they require more energy than the bare minimum. This means that there are devices which are operating at higher power consumption levels, and therefore higher speed. This creates some slack capacity which yields a marginally lower average number of SLA violations.

9.2. *Total response time overflow* ΔR

The total response time overflow gives a measure of "how much" the SLA constraint has been violated. ΔR is defined as the sum, over all times t where $\overline{R}(t) > R_{\max}$, of the difference $\overline{R}(t) - R_{\max}$. In other words, ΔR is the area which lies above the line of the response time constraint $R_{\max}$ and below the curve of measured response time $\overline{R}(\mathbf{S}(t))$ in Fig. 8.

We show in Fig. 10 the comparison of the response time overflow produced by the heuristic and the MIP solver. Again, results in both cases are not statistically different, although average values seem to indicate that the heuristic is marginally better than the MIP solver. As for the average number of SLA violations, this can be explained by considering that configurations produced by the heuristic have higher energy consumption rate, so the heuristic is trading energy for lower response times.

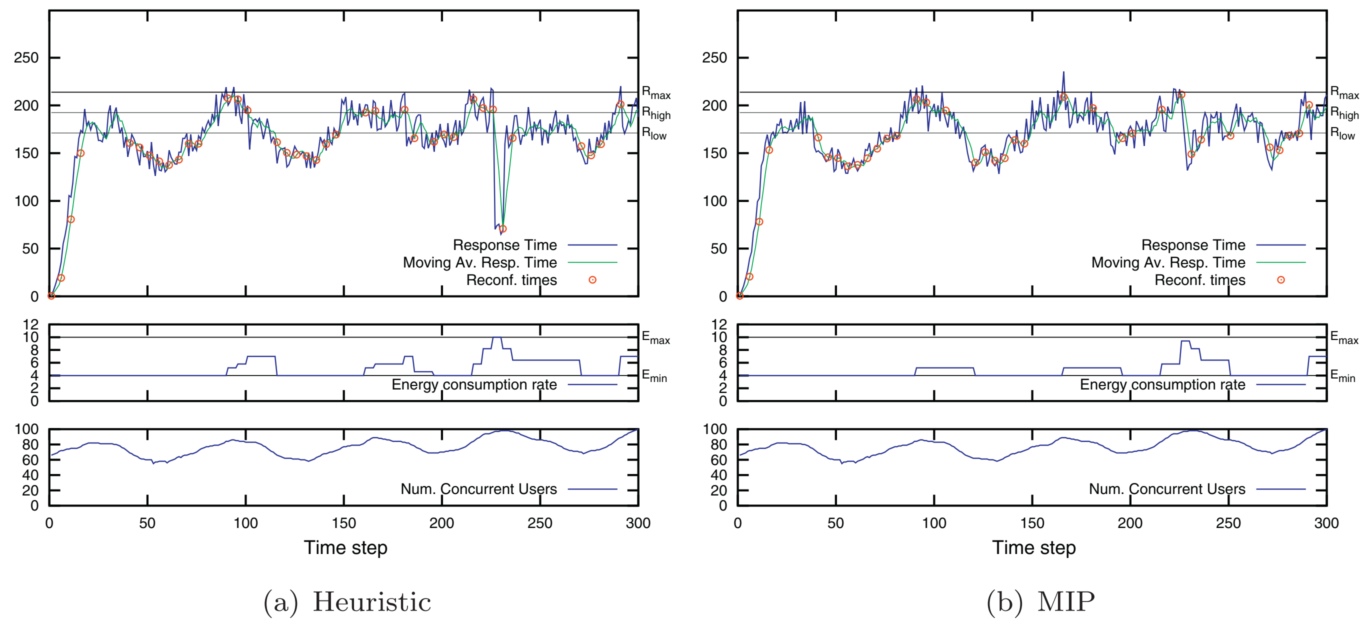

Fig. 8. Results for a single simulation run with $K = 10$ devices and $L = 2$ performance states.

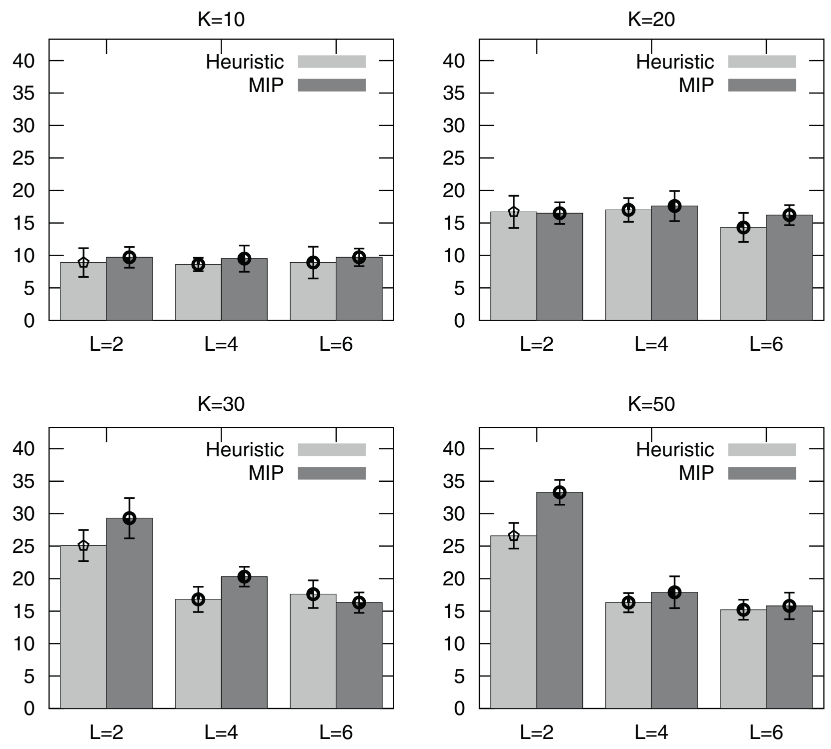

Fig. 9. Number of SLA violations (lower is better).

9.3. *Total energy consumption E_{tot}*

The total energy consumption is defined as the sum of consumption rates $E(\mathbf{S}(t))$ over all time steps $t = 1, \ldots, T$, where $\mathbf{S}(t)$ is the configuration selected by EASY at step t.

Results are shown in Fig. 11; referring to the raw data in Table 6, we observe that the MIP solver produces configurations with lower energy consumption. Those configurations represent the optimal solution of the optimization problem, and therefore are lower bounds of what is possible to achieve with any other heuristic strategy.

9.4. *Execution time*

We measured the total execution time of each simulation run. We remark that the results should not be taken as absolute performance measures, since EASY has been implemented using an interpreted language which is far less efficient than a compiled one. Furthermore, different MIP solvers may have dramatically different convergence speed or may even fail to converge in reasonable time scales on some instances. Therefore, these results should be considered as a qualitative measure of efficiency of the heuristic with respect to the solution of the MIP problem.

From Fig. 12 we see that for small values of K the heuristic described by Algorithms 3 and 4 requires almost the same time as the exact solution of the MIP problem using GLPK. However, as the problem size increases, GLPK is significantly slower than the heuristic, the difference being more than an order of magnitude for $K = 50$. We remark that we forced GLPK to produce a solution within 60 s; for large values of K, GLPK failed in many cases to identify an optimal solution within the allocated time.

9.5. *Total energy overhead E_{ovr}*

This parameter is used to estimate the efficiency of EASY by measuring the fraction of energy, above the minimum consumption rate, which is used by the system. Formally, let $E_{\min}$ be the minimum energy consumption rate when all devices are in state L, and let $E_{\max}$ be the maximum energy consumption rate when all devices are in state 1. Let $E(\mathbf{S}(t))$ be the energy consumption rate

Table 6
Comparison of heuristic and exact solution of the MIP problem.

K	L	MIP				Heuristic			
		SLA viol.	ΔR	E_{tot}	Exec. time	SLA viol.	ΔR	E_{tot}	Exec. time
10	2	9.70 ± 1.59	73.67 ± 14.63	$\underline{1418.70 \pm 13.87}$	4.50 ± 0.04	8.90 ± 2.21	76.29 ± 28.12	1454.70 ± 18.88	$\underline{4.43 \pm 0.03}$
10	4	9.50 ± 2.01	79.50 ± 28.47	$\underline{1277.20 \pm 2.76}$	$\underline{4.64 \pm 0.04}$	8.60 ± 1.03	71.42 ± 16.65	1292.90 ± 3.10	4.73 ± 0.03
10	6	9.70 ± 1.37	64.31 ± 15.39	$\underline{1246.68 \pm 1.89}$	$\underline{4.81 \pm 0.04}$	8.90 ± 2.45	67.75 ± 24.58	1256.88 ± 3.47	5.03 ± 0.03
20	2	16.50 ± 1.67	115.84 ± 14.10	$\underline{3289.80 \pm 31.09}$	$\underline{4.80 \pm 0.06}$	16.70 ± 2.49	137.15 ± 26.16	3569.40 ± 39.91	5.35 ± 0.04
20	4	17.60 ± 2.32	131.35 ± 24.20	$\underline{2740.80 \pm 13.71}$	5.90 ± 0.09	17.00 ± 1.83	140.08 ± 24.86	2822.20 ± 11.28	5.92 ± 0.06
20	6	16.20 ± 1.54	145.47 ± 21.15	$\underline{2627.82 \pm 5.75}$	8.32 ± 0.36	14.30 ± 2.24	116.35 ± 26.47	2667.84 ± 9.26	$\underline{6.62 \pm 0.07}$
30	2	29.30 ± 3.10	244.69 ± 19.78	$\underline{5218.20 \pm 59.86}$	12.01 ± 0.21	25.10 ± 2.40	267.09 ± 48.80	5627.70 ± 27.37	$\underline{6.30 \pm 0.12}$
30	4	20.30 ± 1.52	173.08 ± 39.12	$\underline{4278.10 \pm 10.03}$	54.86 ± 15.98	$\underline{16.80 \pm 1.95}$	172.47 ± 33.57	4386.00 ± 9.39	$\underline{7.52 \pm 0.15}$
30	6	16.30 ± 1.57	186.43 ± 29.05	$\underline{4089.18 \pm 11.03}$	40.47 ± 9.65	17.60 ± 2.12	159.38 ± 30.82	4124.64 ± 5.11	$\underline{8.27 \pm 0.17}$
50	2	33.30 ± 1.91	310.63 ± 41.28	$\underline{8827.80 \pm 38.44}$	195.58 ± 19.93	$\underline{26.60 \pm 1.97}$	246.11 ± 31.73	9517.80 ± 51.00	$\underline{7.61 \pm 0.11}$
50	4	17.90 ± 2.45	208.40 ± 48.19	$\underline{7336.50 \pm 17.71}$	339.11 ± 31.76	16.30 ± 1.50	140.97 ± 27.14	7484.70 ± 10.14	$\underline{9.98 \pm 0.18}$
50	6	15.80 ± 2.04	191.60 ± 59.64	$\underline{6998.70 \pm 7.18}$	146.01 ± 36.48	15.20 ± 1.54	182.25 ± 43.83	7064.34 ± 7.46	$\underline{12.12 \pm 0.25}$

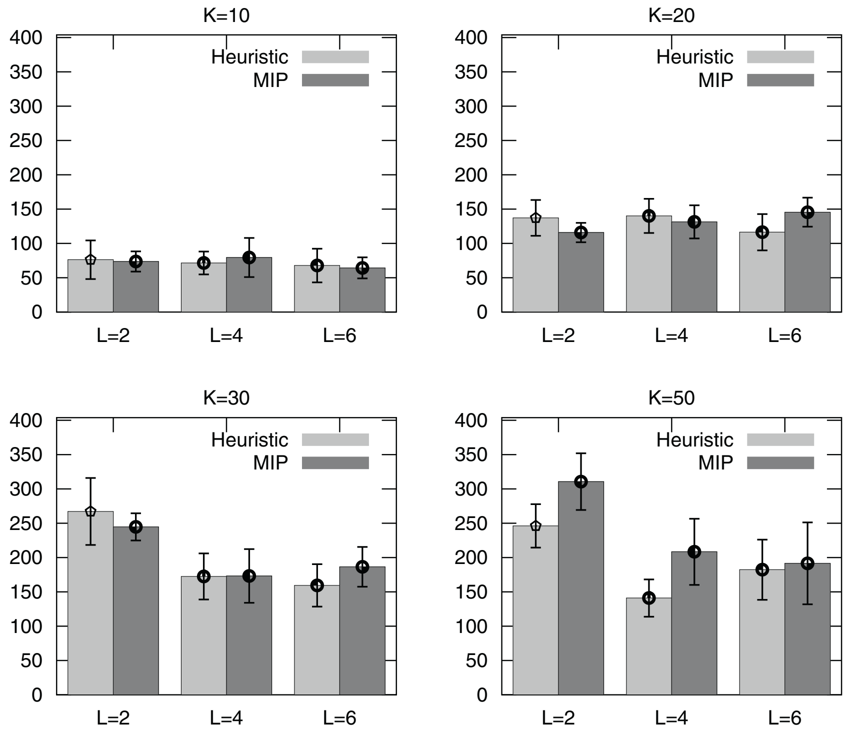

Fig. 10. Response time overflow ΔR (lower is better).

at time t, under the configuration $\mathbf{S}(t)$ produced by EASY. Then, the energy overhead is defined as:

$$E_{\text{ovr}} = \frac{\sum_{t=1}^{T} (E(\mathbf{S}(t)) - E_{\text{min}})}{T \times (E_{\text{max}} - E_{\text{min}})}$$

By definition, the energy overhead is a number in the range [0, 1]. The value $E_{\text{ovr}} = 1$ corresponds to the situation in which all devices are always set in state 0; the value $E_{\text{ovr}} = 0$ corresponds to the situation in which all devices are in state L. Therefore, the energy overhead shows how efficiently the available energy consumption range between E_{min} and E_{max} is being used.

The values of E_{ovr} for all experiments are reported in Tables 4 and 5. From the results we can make an important observation: increasing the number of ACPI performance states L yields a lower value of E_{ovr}. In fact, increasing the number of performance states allows EASY to lower the energy consumption of each device to the minimum level allowing it to deliver

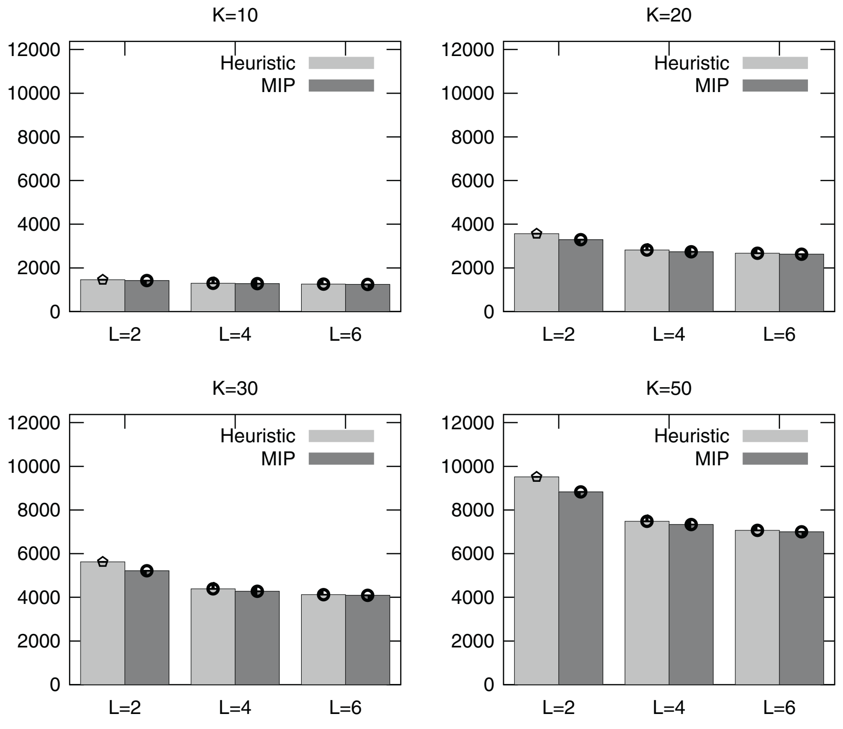

Fig. 11. Total energy consumption E_{tot} (lower is better).

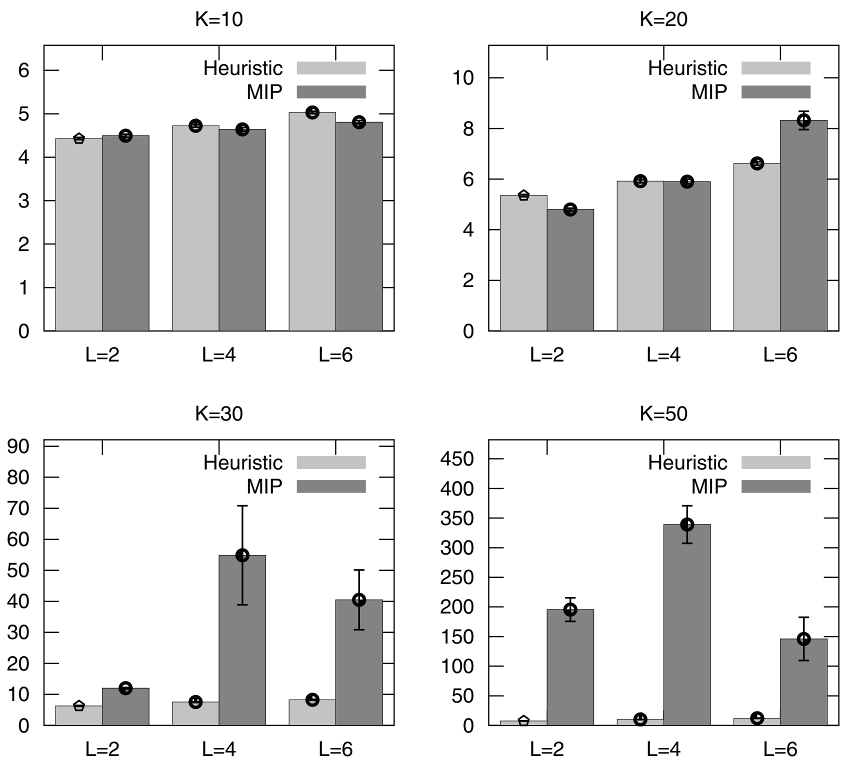

Fig. 12. Average wall-clock execution times (in seconds) of each simulation run (lower is better). Note that each subfigure has different scale.

"just enough" processing power to accommodate its service demand.

10. Conclusions

In this paper we proposed EASY, a framework for reducing the energy consumption rate of large-scale systems with QoS constraints. EASY can be applied to any system made of a set of ACPI-capable devices. The goal of EASY is to set each device to an appropriate ACPI performance state such that (i) the overall system response time is kept below a given threshold, and (ii) the total energy consumption rate is minimized. We formulate the energy-minimization problem as a MIP problem; using a general-purpose MIP solver we can compute exact solutions of the optimization problem, but unfortunately this approach is infeasible even for moderately complex systems due to the large amount of time required to find the solution. Therefore, we proposed an heuristic solution which uses a QN performance model to quickly estimate the response time of a subset of system configurations. We validated the heuristic through numerical experiments, and we observed that the solutions provided by the heuristic are comparable to those obtained by solving the MIP problem; moreover, the heuristic is much faster (more than an order of magnitude) for a system with $K = 50$ devices.

This approach can be extended along different directions. It would be certainly useful adding the possibility to handle non-operating device states (e.g., suspended or powered off) in order to further reduce the power consumption during off peak periods. As described in the introduction, this would pose a set of non-trivial challenges. From one side it would be necessary to migrate data or whole applications away from devices which are being shut down. From the other side it might be appropriate to adopt forecasting techniques in order to predict workload fluctuations ahead of time, in order to have the time to bring machines up or down as requested. Combining EASY with existing energy-conservation approaches based on the use of active/suspended states (e.g., [18]) could be fruitful.

Another direction in which EASY can be extended is that of implementing the control loop shown in Fig. 5 in a totally decentralized way, in order to avoid the possibility that EASY becomes a performance bottleneck or a single point of failure.

Appendix A. Notation

In Table 7 we summarize the main notation adopted throughout the paper.

Appendix B. Queueing network models

Queueing network models [41,40] are a mathematical modeling approach in which a software system is represented as a collection of: (i) *service centers*, which model system resources, and

Table 7
Summary of the notation used in this paper.

$\mathcal{K}$	Index set of devices, $\mathcal{K} = \{1, \ldots, K\}$
$L[k]$	Number of ACPI performance states supported by device $k \in \mathcal{K}$
$\mathcal{L}[k]$	Index set of performance states of device k, $\mathcal{L}[k] = \{1, \ldots, L[k]\}$
$RSP[k, l]$	Relative speed of device k in state $l \in \mathcal{L}[k]$ with respect to the same device in state 0 (fastest)
$EN[k, l]$	Energy consumption rate of device k in state $l \in \mathcal{L}[k]$
$\mathbf{S}, \mathbf{S}'$	System configurations; $\mathbf{S} = (S[1], \ldots, S[K])$, where $S[k] \in \mathcal{L}[k]$ is the current performance state of device k
$\mathbf{S}(t)$	System configuration at time t
$E(\mathbf{S})$	Total energy consumption rate of the system in state $\mathbf{S}$
$\overline{R}(t)$	Measured system response time at time t
$\overline{X}(t)$	Measured system throughput at time t
$\overline{U}[k](t)$	Measured utilization of device $k \in \mathcal{K}$ at time t
$R(\mathbf{S})$	Estimated system response time under configuration $\mathbf{S}$
$D[k]$	Estimated request demanding time of device k
$R_{\max}$	Maximum allowed system response time
R_{high}	Higher threshold, $R_{\text{high}} < R_{\max}$
R_{low}	Lower threshold, $R_{\text{low}} < R_{\text{high}}$

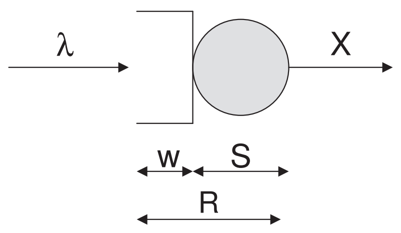

Fig. 13. Single Service Center Model. λ denotes the incoming workload, X denotes the requests throughput. W indicates the requests' waiting time, i.e. the average time spent by requests in the queue.

(ii) *customers*, which represent system users or "requests" and move from one service center to another one. The customers' competition to access resources corresponds to queueing into the service centers.

The simplest queueing network model includes a single service center (see Fig. 13) which is composed of a single queue and a single server: the queue models a flow of customers or requests which enter the system, wait in the queue if the system is busy serving other requests, obtain the service, and then depart. Note that, at any time instant only one customer or request is obtaining the service. Single service centers can be described by two parameters: the requests *arrival rate*, usually denoted by λ, and the average *service time* S, i.e., the average time required by the server in order to execute a single request. The maximum service rate is usually indicated with μ and is defined as $\mu = 1/S$. Given the request arrival rate and the requests service time, QN theory allows evaluating the average value of performance metrics by solving simple equations.

In real systems, requests need to access multiple resources in order to be executed; accordingly, in the model they go through more than a single queue. A QN includes several service centers and can be modeled as a directed graph where each node represents the kth service center, while arcs represent transitions of customers/requests from one service center to the next. The set of nodes and arcs determines the network topology.

Product form QN. One of the most important results of queueing network theory is the *BCMP theorem* (by Baskett, Chandy, Muntz, and Palacios-Gomez) that, under various assumptions (see [47] for further details), shows that performance of a software system is independent of network topology and requests routing but depends only on the requests arrival rate and on the requests *demanding time* $D[k]$, i.e., the average overall time required to complete a request at service center k. The average number of times a request is served at the kth service center is defined as the number of visits $V[k]$, and it holds $D[k] = V[k] \times S[k]$.

In time sharing operating systems, as an example, the average service time is the operating system time slice, while the demanding time is the overall average CPU time required for a request execution. The number of visits is the average number of accesses to the CPU performed by a single request.

Queueing networks satisfying the BCMP theorem assumptions are an important class of models also known as *separable queueing networks* or *product-form models*.[7] The name "separable" comes from the fact that each service center can be separated from the rest of the network, and its solution can be evaluated in isolation. The solution of the entire network can then be formed by combining these separate results [41,47]. Such models are the only ones that can be solved efficiently, while the solution time of the equations governing non-product-form queueing network grows exponentially with the size of the network. Hence, in practical situations the time required for the solution of non-product-form networks becomes prohibitive and approximate solutions have to be adopted.

Open and closed models. Queueing models can be classified as *open* or *closed* models. In open models customers can arrive and depart and the number of customers in the system cannot be determined a priori (and, possibly, it can be infinite). The single service center system in Fig. 13 is an open model. On the other hand, in closed models the number of customers in the system is a constant, i.e., no new customer can enter the system, and no customer can depart. While open models are characterized by the requests arrival rate λ, a closed model can be described by the average number of users in the system N and by their *think time* Z, i.e., the average time that each customer "spends thinking" between interactions (e.g., reading a Web page before clicking the next link). Customers in closed queueing models are represented as delay centers.

Single and multi-class models. Finally, queueing models can be classified as *single-class* and *multi-class* models. In single-class models, customers have the same behavior; on the other hand, in multi-class models, customers behave differently and are mapped into multiple-classes. Each class c can be characterized by different values of demanding time $D[c, k]$ at the kth service center, different arrival rate $\lambda[c]$ in open models, or number of users $N[c]$ and think time $Z[c]$ in closed models. Customers in each class are statistically indistinguishable. Queueing network theory allows determining performance metrics (i.e., response times, utilizations, etc.) on a per-class basis or on an aggregated basis for the whole system.

Solution techniques. After modeling a software system as a queueing network, the model has to be evaluated in order to determine quantitatively the performance metrics.

A first step in the evaluation can be achieved by determining the system bounds; specifically, upper and lower bounds on system throughput and response time can be computed as functions of the system workload intensity (number or arrival rate of customers). Bounds usually require a very little computational effort, especially for the single class case [41].

More accurate results can be achieved by solving the equations governing the QN behavior. Solution techniques can be classified as *analytical methods* (which can be *exact* or *approximate*) and *simulation methods*. Exact analytical methods can determine functional relations between model parameters (i.e., request arrival rate $\lambda[c]$, number of customers $N[c]$ and think time $Z[c]$, and requests demanding times $D[c, k]$) and performance metrics. The analytical solution of open models system is very simple even for multiple class models and yields the average value of the performance metrics. The exact solution of single class closed models is known as the *Mean Value Analysis* (MVA) algorithm and has a linear time complexity with the number of customers and the number of service centers of the models. The MVA algorithm has been extended also to multiple classes, but the time complexity is non-polynomial with the number of customers or with the number of service centers and classes [41]. Hence, large closed models are solved by recurring to approximate solutions, which are mainly iterative methods and can determine approximate results in a reasonable time. Approximate MVA algorithms for multi-class closed models provide results typically within a few percent of the exact analytical solution for throughput and utilization, and within 10% for queue lengths and response time [41].

Analytical solutions can determine the average values of the performance metrics (e.g., average response time, utilization, etc.) or, in some cases, also the percentile distribution of the metric of interest. Determining the percentile distribution of large systems is usually complex even for product-form networks. Indeed, while the mean value of the response time of a request that goes through multiple queues is given by the sum of the average response time

[7] The name "product-form" comes from the fact that the stationary state distribution of the queueing network can be expressed as the product of the distributions of the single queues and avoids the numerical solution of the underlying Markov chain.

obtained locally at the individual queues, the aggregated probability distribution is given by the convolution of the probability distribution of the individual queues. The analytical expression of the percentile distribution becomes complicated for large system (most of the studies provided in the literature are limited to *tandem queues*, i.e., queueing networks including two service centers [48,40]).

Appendix C. MIP problem in GNU MathProg

We describe the MIP model of Fig. 7 using the GNU MathProg language [49], a subset of the AMPL modeling language [50]; the model can be solved using GLPK [51]. Note that in the model below we assume that all devices support the same number `nL` of ACPI performance states; this has been done just for notational convenience, since EASY has been presented in this paper for the general case of heterogeneous devices, and it is also possible to describe the general MIP problem (although less concisely) in AMPL/MathProg.

```
##### Input parameters #########
param N, integer, > 0;                  # Num. of requests
param nK, integer, > 0;                 # Num. of devices
param nL, integer, > 0;                 # Num. of power levels
param Rmax > 0;                         # Max response time

set K:= 1...nK;                         # Devices
set L:= 1...nL;                         # Performance states

# ENE[k,l] is the]energy consumption
# rate of device k in performance state l
param ENE{K, L} > 0;

# DD[k,l] is the estimated service demand
# of device k in performance state l
param DD{K, L} > 0;

##### Variables ################
# X[k,l] = 1 iff device k is
# set at performance level l
var X{K, L}, binary;

var D{K} > = 0;                         # service demands
var Dmax > = 0;                         # max service demand
var Dtot > = 0;                         # sum service demands
var Dave > = 0;                         # average service demand
var R > = 0;                            # response time
var Rplus > = 0;                        # upper bound of R
var Rminus > = 0;                       # lower bound of R

##### Optimization problem #####

# Objective function: minimize power
# consumption
minimize power_consumption:
sum{k in K, l in L} X[k,l]*ENE[k,l];

##### Constraints ##############
# Response time must be less tha Rmax
s.t. response_time: R < = Rmax;

# Each device must be in exactly
# one performance state
s.t. one{k in K}: sum{l in L} X[k,l]=1;

# Define response time
s.t. comp_R: R = (Rplus +Rminus) / 2;

# Define upper bound on response time
s.t. comp_Rplus: Rplus = Dtot +(N-1)*Dmax;

# Define lower bounds on response time
s.t. comp_Rminus1: Rminus > = N*Dmax;
s.t. comp_Rminus2: Rminus > = Dtot +(N-1)*Dave;

# Define maximum service demand
s.t. comp_Dmax {k in K}: Dmax > = D[k];

# Define total service demand
s.t. comp_Dtot: Dtot = sum{k in K} D[k];

# Define average service demand
# (card(K) is the cardinality of set K)
s.t. comp_Dave: Dave = Dtot / card(K);

# Define service demands D[k]
s.t. comp_D{k in K}:
  D[k] = sum{l in L} X[k,l]*DD[k,l];
```

References

[1] J.G. Koomey, Estimating total power consumption by servers in the U.S. and the world (February 5, 2007). http://sites.amd.com/de/Documents/svrpwrusecompletefinal.pdf

[2] R. Das, J.O. Kephart, C. Lefurgy, G. Tesauro, D.W. Levine, H. Chan, Autonomic multi-agent management of power and performance in data centers, in: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Industrial Track, AAMAS'08, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2008, pp. 107–114.

[3] P. Ranganathan, Recipe for efficiency: principles of power-aware computing, Communications of the ACM 53 (4) (2010) 60–67, http://dx.doi.org/10.1145/1721654.1721673.

[4] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, N. Gautam, Managing server energy and operational costs in hosting centers, SIGMETRICS Performance Evaluation Review 33 (1) (2005) 303–314, http://dx.doi.org/10.1145/1071690.1064253.

[5] L.A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37, http://dx.doi.org/10.1109/MC.2007.443.

[6] D. Meisner, C.M. Sadler, L.A. Barroso, W.-D. Weber, T.F. Wenisch, Power management of online data-intensive services, in: Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA'11, ACM, New York, NY, USA, 2011, pp. 319–330, http://dx.doi.org/10.1145/2000064.2000103.

[7] Advanced configuration and power interface specification, revision 4.0a (April 5, 2010). http://www.acpi.info/

[8] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50, http://dx.doi.org/10.1109/MC.2003.1160055.

[9] B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar], Lecture Notes in Computer Science, vol. 5525, Springer, 2009, http://dx.doi.org/10.1007/978-3-642-02161-9.

[10] M.C. Huebscher, J.A. McCann, A survey of autonomic computing-degrees, models, and applications, ACM Computing Surveys 40 (3) (2008) 7:1–7:28, http://dx.doi.org/10.1145/1380584.1380585.

[11] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, Journal of Internet Services and Applications 1 (2010) 7–18, http://dx.doi.org/10.1007/s13174-010-0007-6.

[12] R. Nathuji, K. Schwan, Virtualpower: coordinated power management in virtualized enterprise systems, SIGOPS Operating Systems Review 41 (6) (2007) 265–278, http://dx.doi.org/10.1145/1323293.1294287.

[13] M. Marzolla, O. Babaoglu, F. Panzieri, Server consolidation in clouds through gossiping, in: Int. Symp. World of Wireless, Mobile and Multimedia Networks (WoWMoM), IEEE Computer Society, Lucca, Italy, 2011, pp. 1–6, http://dx.doi.org/10.1109/WoWMoM.2011.5986483.

[14] F. Wuhib, R. Stadler, M. Spreitzer, A gossip protocol for dynamic resource management in large cloud environments, IEEE Transactions on Network and Service Management 9 (2) (2012) 213–225, http://dx.doi.org/10.1109/TNSM.2012.031512.110176.

[15] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, Z. Lu, Power-aware QoS management in web servers, in: RTSS'03: Proceedings of the 24th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 2003, pp. 63–72.

[16] M. Elnozahy, M. Kistler, R. Rajamony, Energy conservation policies for web servers, in: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems – Volume 4, USITS'03, USENIX Association, Berkeley, CA, USA, 2003, p. 8. http://dl.acm.org/citation.cfm?id=1251460.1251468

[17] R. Calinescu, M. Kwiatkowska, Using quantitative analysis to implement autonomic it systems, in: ICSE'09: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2009, pp. 100–110, http://dx.doi.org/10.1109/ICSE.2009.5070512.

[18] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, R. Katz, Napsac: design and implementation of a power-proportional web cluster, SIGCOMM Computer Communication Review 41 (2011) 102–108, http://dx.doi.org/10.1145/1925861.1925878.

[19] Y. Liu, H. Zhu, A survey of the research on power management techniques for high-performance systems, Software: Practice and Experience 40 (11) (2010) 943–964, http://dx.doi.org/10.1002/spe.v40:11.

[20] J.B. Carter, K. Rajamani, Designing energy-efficient servers and data centers, IEEE Computer 43 (7) (2010) 76–78.

[21] D. Barbagallo, E. Di Nitto, D.J. Dubois, R. Mirandola, A bio-inspired algorithm for energy optimization in a self-organizing data center, in: Proceedings of the First International Conference on Self-organizing Architectures, SOAR'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 127–151.

[22] L. Bertini, J.C.B. Leite, D. Mossé, Power optimization for dynamic configuration in heterogeneous web server clusters, Journal of Systems and Software 83 (4) (2010) 585–598, http://dx.doi.org/10.1016/j.jss.2009.10.040.

[23] E.N. Elnozahy, M. Kistler, R. Rajamony, Energy-efficient server clusters, in: PACS'02, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 179–197.
[24] B. Abrahao, V. Almeida, J. Almeida, A. Zhang, D. Beyer, F. Safai, Self-adaptive SLA-driven capacity management for internet services, 2006, pp. 557–568, http://dx.doi.org/10.1109/NOMS.2006.1687584.
[25] I. Cunha, I. Viana, J. Palotti, J. Almeida, V. Almeida, Analyzing security and energy tradeoffs in autonomic capacity management, in: Network Operations and Management Symposium, 2008, NOMS 2008, IEEE, 2008, 2008, pp. 302–309, http://dx.doi.org/10.1109/NOMS.2008.4575148.
[26] G. Jung, M.A. Hiltunen, K.R. Joshi, R.D. Schlichting, C. Pu, Mistral: dynamically managing power, performance, and adaptation cost in cloud infrastructures, in: Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems, ICDCS'10, 2010, pp. 62–73.
[27] N. Huber, F. Brosig, S. Kounev, Model-based self-adaptive resource allocation in virtualized environments, in: SEAMS, ACM, 2011, pp. 90–99.
[28] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, G. Jiang, Power and performance management of virtualized computing environments via lookahead control, Cluster Computing 12 (2009) 1–15.
[29] S. Chen, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, W.H. Sanders, Using CPU gradients for performance-aware energy conservation in multitier systems, Sustainable Computing: Informatics and Systems 1 (2) (2011) 113–133, http://dx.doi.org/10.1016/j.suscom.2011.02.002.
[30] P.J. Denning, J.P. Buzen, The operational analysis of queueing network models, ACM Computing Surveys 10 (3) (1978) 225–261, http://dx.doi.org/10.1145/356733.356735.
[31] J. Zahorjan, K.C. Sevcick, D.L. Eager, B.I. Galler, Balanced job bound analysis of queueing networks, Communications of the ACM 25 (2) (1982) 134–141.
[32] L. Brown, A. Keshavamurthy, D. Shaohua Li, R. Moore, V. Pallipadi, L. Yu, ACPI in linux: architecture, advances and challenges, in: Proceedings of the Linux Symposium – Volume 1, Ottawa, Ontario, Canada, 2004 http://www.linuxsymposium.org/2005/
[33] J.M. Rabaey, A. Chandrakasan, B. Nikolic, Digital Integrated Circuits – A Design Perspective, 2nd ed., Prentice Hall, Upper Saddle River, NJ, USA, 2003.
[34] Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, White Paper (March 2004). ftp://download.intel.com/design/network/papers/30117401.pdf.
[35] AMD PowerNow! Technology. http://amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx
[36] M. Broyles, C. Francois, A. Geissler, M. Hollinger, T. Rosedahl, G. Silva, J.V. Heuklon, B. Veale, IBM EnergyScale for POWER7 Processor-Based Systems, White Paper (August 2010). http://www-03.ibm.com/systems/power/hardware/whitepapers/energyscale7.html
[37] AMD Cool'n'Quiet Technology. http://www.amd.com/us/products/technologies/cool-n-quiet/Pages/cool-n-quiet.aspx
[38] AMD Opteron Processor Power and Thermal Data Sheet, Publication # 30417, Revision 3.11 (May 2006). http://support.amd.com/us/Processor_TechDocs/30417.pdf
[39] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, 2nd ed., Addison-Wesley, Boston, MA, 2005.
[40] G. Bolch, S. Greiner, H. de Meer, K. Trivedi, Queuing Network and Markov Chains, Wiley-Interscience, New York, NY, USA, 1998.
[41] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik, Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Prentice-Hall, Upper Saddle River, NJ, USA, 1984.
[42] J.D.C. Little, A proof for the queuing formula: $L = \lambda W$, Operations Research 9 (3) (1961) 383–387, http://dx.doi.org/10.2307/167570.
[43] M. Reiser, S.S. Lavenberg, Mean-value analysis of closed multichain queuing networks, Journal of the ACM 27 (2) (1980) 313–322.
[44] S.G. Nash, A. Sofer, Linear and Nonlinear Programming, McGraw-Hill, New York, 1996.
[45] J. Lüthi, G. Haring, Mean value analysis for queueing network models with intervals as input parameters, Performance Evaluation 32 (3) (1998) 185–215, http://dx.doi.org/10.1016/S0166-5316(97)00021-7.
[46] J.W. Eaton, GNU Octave Manual, Network Theory Limited, 2002.
[47] F. Baskett, K.M. Chandy, R.R. Muntz, F.G. Palacios, Open, closed, and mixed networks of queues with different classes of customers, Journal of the ACM 22 (2) (1975) 248–260, http://dx.doi.org/10.1145/321879.321887.
[48] R. Jain, The Art of Computer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling, Wiley-Interscience, New York, NY, 1991.
[49] A. Makhorin, Modeling Language GNU MathProg-Language Reference for GLPK version 4.45, draft Edition (December 2010). http://www.gnu.org/software/glpk/glpk.html
[50] R. Fourer, D.M. Gay, B.W. Kernighan, A modeling language for mathematical programming, Management Science 36 (5) (1990) 519–554, http://dx.doi.org/10.1287/mnsc.36.5.519.
[51] A. Makhorin, GNU Linear Programming Kit-Reference Manual for GLPK Version 4.45, draft Edition (December 2010). http://www.gnu.org/software/glpk/glpk.html

Moreno Marzolla graduated in Computer Science from the University of Venezia "Ca' Foscari" (Italy) in 1998, and received a Ph.D. in Computer Science from the same University in 2004. From 2004 to 2005 he was a post-doc researcher at the University of Venezia "Ca' Foscari". From 2005 to 2009 he was a Software Engineer at the Italian National Institute for Nuclear Physics, working in the area of Grid Computing supported by the EGEE, OMII-Europe and EGEE-3 EU-funded projects. In November 2009, he joined the Department of Computer Science of the University of Bologna, where he is currently an assistant professor. His research interests include performance modeling of complex systems and high performance and Cloud computing.

Raffaela Mirandola is Associate Professor in the Dipartimento di Elettronica, Informazione e Bioingegneria at Politecnico di Milano. Raffaela's research interests are in the areas of performance and reliability modeling and analysis of software/hardware systems with special emphasis on: methods for the automatic generation of performance and reliability models for component-based and service-based systems, and methods to develop software that is dependable and can easily evolve, possibly self-adapting its behavior. She has published over 90 journal and conference articles on these topics. She served and is currently serving in the program committees of conferences in the research areas and she is a member of the Editorial Board of Journal of System and Software (published by Elsevier). She has been involved in several national and European research projects among which EU project CASCADAS (IST-027807), Q-ImPreSS (FP7-215013) and SMScom (IDEAS 227077).